Guardian Programmer’s Guide

Abstract

This guide describes how to access operating-system services from an application program using Guardian™ procedure calls.

Product Version

N/A

Supported Release Version Updates

This manual supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.25 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
421922-014	August 2012

Document History

Part Number	Product Version	Published
421922-009	N/A	August 2010
421922-010	N/A	January 2011
421922-011	N/A	August 2011
421922-013	N/A	February 2012
421922-014	N/A	August 2012

Guardian Programmer’s Guide

Glossary Index Figures Tables

What’s New in This Manual xxv
  Manual Information xxv
  New and Changed Information xxv
About This Manual xxix
  Contents xxix
  Related Reading xxx
  Notation Conventions xxxv
Legal Notices xl
  HP Encourages Your Comments xli

1. Introduction to Guardian Programming

Providing Fault Tolerance 1-1
  Application-Level Fault Tolerance 1-2
  Mirrored Disks 1-3
  Multiple Copies of the Operating System 1-3
  System Integrity 1-3
System Services 1-3
  The File System 1-4
  The Startup Sequence 1-7
  Process Management 1-8
  Memory Management 1-8
  Time Management 1-9
  Data Manipulation 1-9
  Debugging, Trap Handling, and Signal Handling 1-9
The Requester/Server Application Model 1-10
  Advantages of the Requester/Server Model 1-11
  Monitoring Server Processes 1-15
  Requesters and Servers in Fault-Tolerant Applications 1-15
  Client/Server Application Model 1-15

1. Introduction to Guardian Programming (continued)

Accessing Guardian Procedures 1-18
  Calling Guardian Procedures From TAL or pTAL 1-18
  Calling Guardian Procedures From C or C++ 1-19
  Calling Guardian Procedures From Pascal 1-20
  Calling Guardian Procedures From COBOL85 1-20
  Calling Guardian Procedures From FORTRAN 1-20
TNS/E Program Execution Modes 1-21
  Similarities and Differences Between H-series RVUs on the TNS/E Platform and G-Series RVUs on the TNS/R Platform 1-21
TNS/R Program Execution Modes 1-23
Using Parameter Declarations Files 1-23
Synchronizing Processes 1-24

2. Using the File System

File Concepts 2-1
  Disk Files 2-1
  Device Files 2-3
  Process Files and $RECEIVE 2-4
File Names 2-4
  Permanent Disk-File Names 2-4
  Temporary Disk-File Names 2-6
  Device-File Names 2-6
  Process File Names 2-7
Location Independent Disk-File Names 2-9
Passing File Names to Processes 2-9
  Using CLASS MAP DEFINEs 2-10
  Using the Startup Sequence 2-10
Creating and Accessing Files 2-10
  Creating Files 2-10
  Opening Files 2-15
  Reading and Writing Data 2-18
  Getting File Information 2-26
  Handling File-System Errors 2-26
  Closing Files 2-29
Accessing Files: An Example 2-29

3. Coordinating Concurrent File Access (continued)

- Setting the Access Mode 3-1
- Setting the Exclusion Mode 3-3
- Locking a File 3-6
- Avoiding Deadlocks 3-7
  - Avoiding Multiple-Process Deadlocks 3-7
  - Avoiding Single-Process Deadlocks 3-10

4. Using Nowait Input/Output

- Overview of Nowait Input/Output 4-1
- Applying a Nowait Operation on a Single File 4-3
- Applying Multiple Nowait Operations on a Single File 4-5
  - Completing I/Os in the Order Initiated 4-6
  - Completing I/Os in Any Order 4-7
  - Using File-System Buffering 4-8
- Applying Nowait Operations to Multiple Files 4-10
- Nowait I/O: An Example 4-12
- Using FILE_COMPLETE_ and its Companion Procedures 4-24
  - Using the FILE_COMPLETE_SET_ Procedure 4-24
  - Using the FILE_COMPLETE_GETINFO_ Procedure 4-27
  - Using the FILE_COMPLETE_ Procedure 4-27
- Nowait-Depth 4-30

5. Communicating With Disk Files

- Types of Disk Files 5-1
  - Unstructured Files 5-1
  - Structured Files 5-2
  - Alternate-Key Files 5-7
  - Queue Files 5-8
- Using Unstructured Files 5-9
  - Creating Unstructured Files 5-9
  - Opening Unstructured Files 5-11
  - Positioning, Reading, and Writing With Unstructured Files 5-11
  - Locking With Unstructured Files 5-11
  - Renaming Unstructured Files 5-12
  - Avoiding Unnecessary Cache Flushes to Unstructured Files 5-13
  - Closing Unstructured Files 5-13
  - Purging Unstructured Files 5-14
  - Altering Unstructured-File Attributes 5-15

5. Communicating With Disk Files (continued)

- Using Relative Files 5-15
  - Creating Relative Files 5-16
  - Opening Relative Files 5-17
  - Positioning, Reading, and Writing With Relative Files 5-17
  - Locking, Renaming, Caching, Closing, Purging, and Altering Relative Files 5-18
  - Relative-File Programming Example 5-18
- Using Entry-Sequenced Files 5-31
  - Creating Entry-Sequenced Files 5-31
  - Opening Entry-Sequenced Files 5-32
  - Positioning, Reading, and Writing With Entry-Sequenced Files 5-32
  - Locking, Renaming, Caching, Closing, Purging, and Altering Entry-Sequenced Files 5-33
  - Monitoring Writes to a Disk File 5-33
  - Entry-Sequenced File Programming Example 5-36
- Using Key-Sequenced Files 5-47
  - Creating Key-Sequenced Files 5-47
  - Opening Key-Sequenced Files 5-48
  - Positioning, Reading, and Writing With Key-Sequenced Files 5-49
  - Locking, Renaming, Caching, Closing, Purging, and Altering Key-Sequenced Files 5-51
  - Key-Sequenced File Programming Example 5-51
  - Using Alternate Keys With an Entry-Sequenced File 5-72
  - Using Alternate Keys With a Key-Sequenced File 5-72
- Using Partitioned Files 5-77
  - Creating Partitioned Files 5-77
  - Accessing Partitioned Files 5-80
- Using Alternate Keys 5-81
  - Creating Alternate-Key Files 5-81
  - Adding Keys to an Alternate-Key File 5-85
  - Using Alternate Keys With a Relative File 5-86

6. Communicating With Processes

- Sending and Receiving Messages: An Introduction 6-1
- Sending Messages to Other Processes 6-4
  - Opening a Process 6-4
  - Writing Messages to Another Process 6-6
- Queuing Messages on $RECEIVE 6-8

6. Communicating With Processes (continued)

- Receiving and Replying to Messages From Other Processes 6-10
  - Opening $RECEIVE for Two-Way Communication 6-10
  - Reading Messages for Two-Way Communication 6-11
  - Replying to Messages 6-11
  - Sending, Receiving, and Replying to Messages: An Example 6-13
  - Closing $RECEIVE 6-13
- Receiving Messages From Other Processes: One-Way Communication 6-15
  - Opening $RECEIVE for One-Way Communication 6-15
  - Reading From $RECEIVE for One-Way Communication 6-15
  - Sending and Receiving One-Way Messages: An Example 6-15
- Handling Multiple Messages Concurrently 6-17
  - Opening $RECEIVE to Allow Concurrent Message Processing 6-17
  - Reading Messages for Concurrent Processing 6-18
  - Getting Information About Messages Read From $RECEIVE 6-19
  - Replying to Messages 6-21
  - Handling Multiple Messages Concurrently: An Example 6-22
- Checking for Canceled Messages 6-23
  - Checking for Cancellation Messages 6-24
  - Using the MESSAGESTATUS Procedure 6-25
- Receiving and Processing System Messages 6-26
  - Receiving System Messages 6-27
  - Processing Open and Close System Messages 6-29
  - Processing Control, Setmode, Setparam, and Controlbuf Messages 6-30
- Handling Errors 6-31
- Communicating With Processes: Sample Programs 6-31
  - Programming the Requester 6-32
  - Programming the Server 6-51

7. Using DEFINEs

- Example Uses for DEFINEs 7-2
  - Example of a CLASS MAP DEFINE 7-3
  - Example of a CLASS SEARCH DEFINE 7-4
  - Example of a CLASS TAPE DEFINE 7-5
  - CLASS DEFAULTS DEFINEs 7-5
- DEFINE Names 7-7

7. Using DEFINEs (continued)

- DEFINE Attributes 7-8
  - Attribute Data Types 7-8
  - Attribute Values 7-9
  - CLASS Attribute 7-9
- Working With DEFINEs 7-10
  - Enabling DEFINEs 7-10
  - Referring to DEFINEs 7-11
- Adding DEFINEs 7-11
  - Setting Attributes in the Working Set 7-12
  - Checking the Working Set for Errors 7-13
  - Adding a DEFINE to the Context of Your Process 7-14
  - Deleting DEFINEs From the Process Context 7-15
  - Saving and Restoring DEFINEs 7-15
  - Saving and Restoring the Working Set 7-16
- Using DEFINEs: An Example 7-17

8. Communicating With a TACL Process

- Setting Up the Process Environment 8-1
- Obtaining Startup Information 8-3
  - Using INITIALIZER to Read the Startup Message 8-6
  - Processing the Startup Message 8-6
- Using ASSIGNs and PARAMs 8-8
  - Using INITIALIZER to Read Assign and Param Messages 8-12
  - Processing Assign Messages 8-12
  - Processing the Param Message 8-14
- Setting a Timeout Value for INITIALIZER 8-16
- Reading the Startup Sequence Without INITIALIZER 8-16
- Waking the TACL Process 8-19
- Causing the TACL Process to Display Text 8-20

9. Communicating With Devices

- Overview of I/O Subsystem 9-1
- Addressing Devices 9-2
- Accessing Devices 9-3
- Controlling Devices 9-4

9. Communicating With Devices (continued)

Getting Device Information 9-5
  Additional Device Information (G-series Only) 9-8

10. Communicating With Terminals

Accessing a Terminal 10-1
  Opening a Terminal 10-4
  Transferring Data Between Application and Terminal 10-6
  Timing Out Terminal Response 10-8
  Echoing Text to the Terminal 10-9
  Setting the Transfer Mode 10-9
  Terminating Terminal Access 10-10
Communicating in Conversational Mode 10-10
  Using the Line-Termination Character 10-11
  Setting the Interrupt Characters for Conversational Mode 10-12
  Controlling Forms Movement 10-16
Communicating in Page Mode 10-18
  Using the Page-Termination Character 10-19
  Setting the Interrupt Characters for Page Mode 10-20
  Communicating With Pseudopolled Terminals 10-22
Managing the BREAK Key 10-23
  Taking BREAK Ownership 10-25
  Releasing BREAK Ownership 10-26
  Selecting BREAK Mode 10-27
Recovering From Errors 10-36
  Recovering From Errors That Indicate a Temporary Lack of Resources 10-37
  Recovering From an “Operation Timed Out” Error 10-37
  Recovering From a BREAK Error 10-37
  Responding to Operator Preemption 10-38
  Recovering From a Modem Error 10-38
  Recovering From a Path Error 10-38
  Recovering From Errors: A Sample Program 10-39

11. Communicating With Printers

Accessing a Printer 11-2
  Procedures for Working With Printers 11-2
  A Printer Program Outline 11-4

11. Communicating With Printers (continued)

- Using the Printer Control Language 11-5
  - Controlling the Printer 11-6
  - Commonly Used PCL Escape Sequences 11-8
- Programming for Tandem Laser Printers 11-9
  - Selecting a Printer Language (5577 Only) 11-10
  - Using Job-Control Commands 11-11
  - Using Page-Control Commands 11-14
  - Printing Text 11-16
  - Resetting the Laser Printer Default Values 11-19
- Programming for Tandem Matrix Line Printers 11-19
  - Using Page-Control Commands 11-20
  - Controlling Forms Movement 11-21
  - Printing Text 11-26
  - Resetting the Printer to Default Values 11-29
- Recovering From Errors 11-29
  - Recovering From a “Device Not Ready” Error 11-30
  - Recovering From Path Errors 11-30
- Sample Program for Using a Printer 11-31

12. Communicating With Magnetic Tape

- Accessing Magnetic Tape: An Introduction 12-2
- Positioning the Tape 12-4
  - Spacing Forward and Backward by Files 12-5
  - Spacing Forward and Backward by Record Blocks 12-7
  - Rewinding the Tape 12-10
- Reading and Writing Tape Records 12-11
  - Reading Tape Records 12-11
  - Writing Tape Records 12-13
- Blocking Tape Records 12-15
- Working in Buffered Mode 12-16
  - Invoking and Revoking Buffered-Mode Operation 12-17
  - Flushing the Buffer 12-18
  - Buffered Mode for Streaming Devices (D-Series Only) 12-19
  - Buffering End-of-File Marks 12-19
  - An Example of Buffered-Mode Operation 12-19

12. Communicating With Magnetic Tape (continued)

- Working With Standard Labeled Tapes 12-21
  - Enabling Labeled Tape Processing 12-21
  - Creating Labeled Tapes 12-21
  - Checking for Labeled Tape Support 12-21
  - Accessing Labeled Tapes 12-22
  - Writing to the Only File on a Labeled Tape Volume 12-29
  - Writing to a File on a Multiple-File Labeled Tape Volume 12-33
  - Writing to a File on Multiple Labeled Tape Volumes 12-38
  - Reading From the Only File on a Labeled Tape Volume 12-42
  - Reading From a File on a Multiple-File Labeled Tape Volume 12-45
  - Reading From a File on Multiple Labeled Tape Volumes 12-48
- Accessing a Labeled Tape File: An Example 12-52
  - Preparing the Tape 12-52
  - Creating the DEFINE 12-52
  - Writing the Program 12-53
- Working With Unlabeled Tapes 12-70
  - Accessing Unlabeled Tapes 12-70
  - Writing to a Single-File Unlabeled Tape 12-74
  - Writing to a Multiple-File Unlabeled Tape 12-77
  - Writing to a File on Multiple Unlabeled Tape Reels 12-79
  - Reading From a Single-File Unlabeled Tape 12-82
  - Reading From a Multiple-File Unlabeled Tape 12-83
  - Reading From a File on Multiple Unlabeled Tape Reels 12-84
- Terminating Tape Access 12-84
- Recovering From Errors 12-85
  - Recovering From “Device Not Ready” Errors 12-87
  - Recovering From Tape Unit Power Failure 12-87
  - Recovering From Path Errors 12-88
- Accessing an Unlabeled Tape File: An Example 12-89

13. Manipulating File Names

- Overview 13-1
  - Identifying Portions of File Names 13-2
  - Working With File-Name Patterns 13-3
- Scanning, Resolving, and Unresolving File Names 13-4
  - Scanning a String for a Valid File Name 13-4
  - Resolving Names 13-7
  - Truncating Default Parts of File Names 13-13
  - Extracting Pieces of File Names 13-15
- Modifying Portions of a File Name 13-18
  - Modifying One Part of a File Name 13-19
  - Replacing a File-Name Suffix or File-Name Prefix 13-19
  - Replacing a Subpart of a Process ID 13-20
- Comparing File Names 13-20
- Searching For and Matching File-Name Patterns 13-21
  - Establishing the Start of a File-Name Search 13-21
  - Finding the Next Matching File Name 13-26
  - Terminating the File-Name Search 13-31
  - File-Name Matching 13-31
- Manipulating File Names: An Example 13-33

14. Using the IOEdit Procedures

- When to Use and When Not to Use EDIT Files 14-1
- Overview of IOEdit 14-2
  - When Should You Use IOEdit? 14-2
  - Line Numbers and Records 14-4
  - Packed Line Format 14-5
  - The EDIT File Segment 14-7
  - IOEdit and Errors 14-7
- Creating, Opening, and Initializing an IOEdit File 14-8
  - Opening an Already Existing File 14-8
  - Opening a Nonexistent File 14-9
  - Initializing an Already Open File 14-10

14. Using the IOEdit Procedures (continued)

- Reading and Writing an IOEdit File 14-11
  - Record Pointers 14-11
  - Selecting a Starting Point 14-12
  - Performing Sequential Reading 14-13
  - Performing Sequential Writing 14-13
  - Setting and Getting the Record Number Increment 14-15
  - Renumbering Lines 14-15
  - Handling “File Full” Errors 14-16
  - Deleting Lines 14-16
  - Line Backspacing 14-17
- Using Nowait I/O With IOEdit Files 14-18
- Compressing an IOEdit File 14-18
- Closing an IOEdit File 14-19
  - Closing a Single File 14-19
  - Closing All EDIT Files 14-19

15. Using the Sequential Input/Output Procedures

- An Introduction to the SIO Procedures 15-2
  - FCBs for SIO Files 15-2
  - Steps for Writing a Program 15-3
  - Differences Between TNS/R Native and TNS Procedures 15-3
- Initializing SIO Files Using TAL or pTAL DEFINEs 15-5
  - Setting Up the SIO Data Structures 15-6
  - Assigning a Logical File Name 15-10
  - Using the INITIALIZER Procedure 15-12
  - Setting Up File Access 15-14
  - Reassigning a Physical File Name to a Logical File 15-19
  - Sample Initialization 15-19
- Opening and Creating SIO Files 15-23
  - Setting Flag Values in the OPEN^FILE Call 15-23
  - Opening SIO Files: Simplest Form 15-24
  - Creating SIO Files 15-24
  - Block Buffering With SIO Files 15-25
  - Purging Data When Opening 15-26
- Getting Information About SIO Files 15-26

15. Using the Sequential Input/Output Procedures (continued)

- Reading and Writing SIO Files 15-27
  - Handling Basic I/O With SIO Files 15-28
  - Changing the Interactive Read Prompt 15-29
  - Handling Long Writes 15-30
  - Handling Padding Characters 15-31
  - Writing to a Printer 15-36
- Accessing EDIT Files 15-37
  - Opening an EDIT File 15-38
  - Setting the Read Position 15-38
- Handling Nowait I/O 15-39
  - Waiting for One File 15-39
  - Waiting for Any File 15-42
- Handling Interprocess Messages 15-45
  - Passing Messages and Reply Text Between Processes 15-45
  - Passing Messages Between Processes: No Reply Data 15-48
  - Communicating With Multiple Processes 15-52
- Handling System Messages 15-52
  - Selecting or Masking System Messages 15-53
  - Reading System Messages 15-54
- Handling BREAK Ownership 15-54
  - Taking BREAK Ownership 15-55
  - Checking for a Break Message 15-56
  - Returning BREAK Ownership 15-56
  - Handling BREAK Ownership: An Example 15-57
  - Handling BREAK Ownership With $RECEIVE Handled as a Non-SIO File 15-59
- Handling SIO Errors 15-60
  - Handling Error Messages 15-60
  - Handling Fatal Errors 15-61
  - Handling Retryable Errors 15-63
- Closing SIO Files 15-65
- Initializing SIO Files Without TAL or pTAL DEFINEs 15-65
  - Allocating FCBs 15-66
  - Initializing FCBs 15-67
  - Naming FCBs 15-67
  - Setting Up File Access Without INITIALIZER 15-67
  - Sample Initialization 15-68
- Using the SIO Procedures: An Example 15-71

16. Creating and Managing Processes

- Process Management Overview 16-1
  - Process Identifiers 16-2
  - Programs and Processes 16-3
  - Process Organization 16-5
  - Process Security 16-10
  - Relationship With Other Processes 16-11
  - Relationship With a Home Terminal 16-14
  - Process Subtype 16-15
  - Process Priority 16-15
  - Process Execution 16-18
- Creating Processes 16-19
  - Using the PROCESS_LAUNCH_ Procedure 16-21
  - Creating an Unnamed Process 16-23
  - Creating a Named Process 16-23
  - Creating a Process in a Nowait Manner 16-26
  - Analyzing Process-Creation Errors 16-27
  - Specifying Process Attributes and Resources 16-28
- Sending the Startup Sequence to a Process 16-38
  - Sending and Receiving the Startup Message 16-39
  - Sending and Receiving Assign and Param Messages 16-46
- Monitoring a Child Process 16-46
- Deleting Processes 16-47
  - Deleting Your Own Process 16-48
  - Deleting Other Processes 16-50
  - Using Stop Mode to Control Process Deletion 16-51
  - Reusing Resources Held by a Stopped Process 16-51
- Suspending and Activating Processes 16-52
  - Suspending Your Own Process 16-52
  - Suspending Other Processes 16-52
  - Activating Another Process 16-53
- Getting and Setting Process Information 16-53
  - Getting Process Information 16-53
  - Setting Process Information 16-60
- Manipulating Process Identifiers 16-61
  - Retrieving Information From a Process Handle 16-62
  - Converting Between Process Handles and Process File Names 16-62
  - Controlling the IPU Affinity of Processes 16-64

17. Managing Memory

An Introduction to Memory-Management Procedures 17-1
Managing the User Data Areas 17-2
  Managing the TNS User Data Segment 17-2
  Managing the Native User Data Areas 17-6
  Checking the Bounds of Your Data Areas 17-15
Using (Extended) Data Segments 17-16
  Overview of Selectable Segments 17-17
  Overview of Flat Segments 17-19
  Which Type of Segment Should You Use? 17-20
  Using Selectable Segments in TNS Processes 17-21
  Accessing Data in Extended Data Segments 17-22
  Attributes of Extended Data Segments 17-23
  Allocating Extended Data Segments 17-23
  Checking Whether an Extended Data Segment Is Selectable or Flat 17-28
  Making a Selectable Segment Current 17-28
  Referencing Data in an Extended Data Segment 17-29
  Checking the Size of an Extended Data Segment 17-34
  Changing the Size of an Extended Data Segment 17-34
  Transferring Data Between an Extended Data Segment and a File 17-35
  Moving Data Between Extended Data Segments 17-37
  Checking Address Limits of an Extended Data Segment 17-38
  Sharing an Extended Data Segment 17-39
  Determining the Starting Address of a Flat Segment 17-41
  Deallocating an Extended Data Segment 17-42
Using Memory Pools 17-42
  Defining a Memory Pool 17-43
  Getting Space in a Memory Pool 17-46
  Returning Memory Pool Space 17-47
  Changing the Size of a Memory Pool 17-47
  Getting Information About a Memory Pool 17-48
  Debugging a Memory Pool 17-48

18. Managing Time

How the System Keeps Time 18-1
  Clock Averaging and System Time 18-1
  Time Zones and Daylight Saving Time 18-2
  128-Bit, 64-Bit, and 48-Bit Timestamps 18-2

18. Managing Time (continued)

Using the Time Management Procedures 18-4
Time and Date Manipulation 18-6
  Working With 64-Bit Julian Timestamps 18-6
  Working With Julian Day Numbers 18-12
  Working With 48-Bit Timestamps 18-13
Timing in Elapsed Time and Timing in Process Time 18-15
  Setting and Canceling Timers: Elapsed Time 18-16
  Setting and Canceling Timers: Process Time 18-17
  Timing Your Process 18-19
  Timing Another Process 18-19
  Converting Process Time Into a Readable Form 18-19
Measuring Long Time Intervals 18-20
  A Sample Long-Range Timer 18-20
Managing System Time 18-25
  Checking the System Clock 18-26
  Setting the System Clock 18-26
  Interacting With the DST Transition Table 18-29

19. Formatting and Manipulating Character Data

Using the Formatter 19-1
  Format-Directed Formatting 19-2
  List-Directed Formatting 19-25
Manipulating Character Strings 19-35
  Converting Between Strings and Integers 19-35
  Case Shifting Character Strings 19-36
  Editing a Character String 19-37
  Sorting Characters 19-46
Programming With Multibyte Character Sets 19-49
  Checking for Multibyte Character-Set Support 19-51
  Determining the Default Character Set 19-52
  Analyzing a Multibyte Character String 19-52
  Dealing With Fragments of Multibyte Characters 19-54
  Handling Multibyte Blank Characters 19-54
  Determining the Character Size of a Multibyte Character Set 19-54
  Case Shifting With Multibyte Characters 19-55
  Testing for Special Symbols 19-55
  Sample Program 19-56

20. Interfacing With the ERROR Program

Creating an ERROR Process 20-2
Opening an ERROR Process 20-3
Sending an ERROR Process a Startup Message 20-4
Reading and Processing Error-Message Text 20-5
Closing and Deleting an ERROR Process 20-5
Using the ERROR Process: An Example 20-6

21. Writing a Requester Program

Functions of a Requester 21-1
  Terminal Interface 21-2
  Field Validation 21-2
  Data Mapping 21-2
  Application Control 21-3
File System I/O Synchronization 21-4
  Sync-Depth 21-5
  Sync-Depth in Practice 21-6
  File Sync Block Checkpoints (Example) 21-7
Writing a Requester Program: An Example 21-9
  User Interface 21-10
  Application Overview 21-10
  Coding the Requester Program 21-14

22. Writing a Server Program

Functions of a Server Process 22-1
  Multithreaded and Single-Threaded Servers 22-1
  Receive-Depth 22-2
  Context-Free Servers 22-3
Maintaining an Opener Table 22-3
  The Opener Table 22-3
  Getting Message Information 22-4
  Adding a Requester to the Opener Table 22-5
  Checking a Request Against the Opener Table 22-6
  Deleting a Requester From the Opener Table 22-6
Writing a Server Program: An Example 22-8
  Application Overview 22-8
  The Part-Query Server ($SER1) 22-10
  The Process-Order Server ($SER2) 22-28
  The Order-Query Server ($SER3) 22-48

23. Writing a Command-Interpreter Monitor ($CMON)

Communicating With TACL Processes 23-2
Controlling the Configuration of a TACL Process 23-4
    Retaining Default Values 23-7
    Setting Configuration Parameters 23-7
Controlling Logon and Logoff 23-8
    Controlling Logon 23-9
    Controlling Logoff 23-12
    Controlling Illegal Logon 23-13
Controlling Passwords 23-14
    When the User Requests to Change a Local Password 23-14
    When the User Requests to Change a Remote Password 23-15
Controlling Process Creation 23-17
    Controlling the Priority of a New Process 23-20
    Controlling the CPU of a New Process 23-21
Controlling Change of Process Priority 23-22
Controlling Adding and Deleting Users 23-24
    Controlling Adding a User 23-24
    Controlling Deleting a User 23-26
Controlling $CMON While the System Is Running 23-27
    Setting the Logon Display Text at Run Time 23-29
    Refusing Command-Interpreter Requests 23-31
    Controlling Which CPU a Process Can Run In 23-33
Writing a $CMON Program: An Example 23-36
    Sample $CMON Program 23-36
    Sample Command-Interface Program 23-70
Debugging a TACL Monitor ($CMON) 23-89
    A TACL Macro for Debugging and Testing a $CMON Program 23-89
    Procedure for Debugging and Testing a TACL Monitor ($CMON) 23-92

24. Writing a Terminal Simulator

Specifying Device Subtype 30 24-2
    Why Device Subtype 30 Must Be Specified 24-2
    How to Specify Device Subtype 30 24-2
Assigning a Name to the Terminal-Simulation Process 24-3
Accepting System Messages Through $RECEIVE 24-3

24. Writing a Terminal Simulator (continued)

Specifying How to Process System Messages 24-4
  Allowing the Requester to Specify the *`last-params`* Parameter 24-4
  Allowing the Requester to Call SETPARAM 24-5
Processing I/O Requests 24-5
Processing System Messages 24-9
  Processing Control Messages 24-10
  Processing Setmode Messages 24-10
  Processing Setparam Messages 24-11
  Processing Device-Type Information Requests 24-12
Managing the BREAK Key 24-13
  Tracking the BREAK Owner 24-14
  Basing Interprocess I/O on BREAK Mode 24-14
  Sending Break-on-Device Messages 24-15

25. Debugging, Trap Handling, and Signal Handling

Invoking a Debugger 25-1
  Getting a Process Into the Debug State 25-2
  Specifying the Debugging Environment 25-6
Handling Trap Conditions 25-9
  Setting Up a Trap Handler 25-11
  Processing a Trap 25-12
  Exiting a Trap Handler 25-13
  Disabling Trap Handling 25-14
  Trap Handling on Native Systems 25-15
  Writing a Trap Handler: Examples 25-16
Handling Signals 25-20
  About Signals 25-21
  Comparing Traps and Signals 25-22
  When Would You Use a Signal Handler? 25-23
  Standard Signals Functions 25-23
  Using Standard Signals Functions 25-26
  HP Extensions 25-27
  Using HP Extensions 25-28
  Interoperability Considerations 25-29
  Examples 25-29

26. Synchronizing Processes

How Binary Semaphores Work 26-2
Summary of Guardian Binary Semaphore Procedures 26-4
Using the Binary Semaphore Procedure Calls 26-5
  Creating a Binary Semaphore 26-5
  Opening a Binary Semaphore 26-6
  Locking a Binary Semaphore 26-6
  Unlocking a Binary Semaphore 26-7
  Testing Ownership of a Binary Semaphore 26-7
  Forcing a Lock on a Binary Semaphore 26-8
  Closing a Binary Semaphore 26-8
Binary Semaphore Interface Declarations 26-8
Binary Semaphore Example 26-10
  Shared Structure 26-10
  External Declarations 26-10
  Procedure USERESOURCE 26-11
  Procedure PRIMARY 26-11
  Procedure SECONDARY 26-12
BINSEM_GETSTATS_ and BINSEM_STAT_VERSION_ Example 26-12

27. Fault-Tolerant Programming in C

Overview of Active Backup Programming 27-1
Summary of Active Backup Processing 27-2
What the Programmer Must Do 27-3
  Planning Tasks 27-3
  Programming Tasks 27-4
C Extensions That Support Active Backup Programming 27-5
  Starting the Backup Process 27-5
  Opening a File With a Specified Sync Depth 27-6
  Retrieving File Open State Information in the Primary Process 27-6
  Opening Files in the Backup Process 27-6
  Retrieving File State Information in the Primary Process 27-7
  Updating File State Information in the Backup Process 27-7
  Terminating the Primary and Backup Processes 27-7
Organizing an Active Backup Program 27-8
  Primary Process Organization 27-9
  Backup Process Organization 27-10

27. Fault-Tolerant Programming in C (continued)

- Updating State Information 27-11
  - Types of State Information 27-13
  - Updating Control State Information 27-14
  - Updating File State Information 27-15
  - Updating Application State Information 27-18
  - Guidelines for Updating State Information 27-18
  - Example of Updating State Information 27-20
  - Saving State Information for Multiple Disk Updates 27-23
- Providing Communication Between the Primary and Backup Processes 27-23
  - Sending Messages From the Primary to the Backup 27-24
  - Receiving Messages in the Backup Process 27-25
  - Monitoring the Backup Process 27-26
- Programming Considerations 27-27
  - Compile-Time and Linker Considerations 27-27
  - Run-Time Considerations 27-28
- Comparison of Active Backup and Passive Backup 27-29
- Active Backup Example 1 27-32
  - Program Declarations 27-35
  - Creating and Starting the Backup Process 27-39
  - Updating State Information 27-41
  - Primary and Backup Processing 27-43
  - Compiling and Running the Example 27-48
  - Example With Debugging Options 27-48
- Active Backup Example 2 27-52
  - Program Declarations 27-53
  - Creating and Starting the Backup Process 27-58
  - Updating State Information 27-58
  - Primary and Backup Processing 27-59
  - Compiling and Running the Example 27-65

28. Using Floating-Point Formats

- Differences Between Tandem and IEEE Floating-Point Formats 28-1
- Building and Running IEEE Floating-Point Programs 28-2
- Compiling and Linking Floating-Point Programs 28-2
- Link-Time Consistency Checking 28-3
- Run-Time Consistency Checking 28-4
- Run-Time Support 28-4
- Debugging Options 28-4

28. Using Floating-Point Formats (continued)

Conversion Routines 28-5
Floating-Point Operating Mode Routines 28-5

A. Mixed Data Model Programming

Using 64-bit Addressable Memory A-1
  Accessing Data in 64-bit Segments A-1
  Allocating a 64-bit Segment A-2
  Dynamic Memory Allocation in 64-bit Segments A-3
  Data Scanning and Movement within 64-bit Segments A-4
  File I/O to/from 64-bit Segments A-4
  Socket I/O to/from 64-bit Segments A-5
  OSS I/O to/from 64-bit segments A-5
  Debugging Programs with 64-bit Segments A-6
  Examples A-6

Glossary

Index

Figures

Figure i. Related Manuals xxxii
Figure 1-1. A Requester/Server Application 1-10
Figure 1-2. Multiple Users in a Requester/Server Application 1-12
Figure 1-3. Multiple Functions in a Requester/Server Application 1-13
Figure 1-4. Requester/Server Application in a Network Environment 1-14
Figure 1-5. Client/Server Architecture 1-16
Figure 1-6. Distributed Client/Server 1-17
Figure 2-1. Disk Files 2-3
Figure 2-2. File Space Allocated in Extents 2-12
Figure 3-1. Exclusion and Access Mode Compatibility 3-5
Figure 3-2. Two Processes in Deadlock 3-8
Figure 3-3. Avoiding the Two-Process Deadlock 3-9
Figure 3-4. Single-Process Deadlock 3-10
Figure 3-5. Avoiding the Single-Process Deadlock 3-11
Figure 4-1. Waited and Nowait I/O 4-2
Figure 4-2. Multiple Concurrent Operations on One File 4-5
Figure 4-3. Nowait Operations on Multiple Files 4-10
Figure 5-1. Relative-File Structure 5-3
Figure 5-2. Entry-Sequenced File Structure 5-4

Figure 5-3. Key-Sequenced File Structure 5-6
Figure 5-4. An Alternate-Key File 5-8
Figure 5-5. Monitoring Write Operations on a Disk File 5-34
Figure 5-6. Example of Alternate-Key File for Use With an Entry-Sequenced File 5-72
Figure 5-7. Example of Alternate-Key File for Use With a Key-Sequenced File 5-73
Figure 5-8. Sample Partitioned File 5-80
Figure 5-9. Example of Alternate-Key File for Use With a Relative File 5-86
Figure 6-1. Sending and Receiving Messages 6-2
Figure 6-2. Multiple Requester Processes and Message Queuing 6-8
Figure 6-3. Message Queuing by Process Priority 6-9
Figure 6-4. Two-Way Interprocess Communication 6-14
Figure 6-5. One-Way Interprocess Communication 6-16
Figure 6-6. Example of Handling Multiple Messages Concurrently 6-22
Figure 6-7. Example of a Requester/Server Application 6-31
Figure 9-1. Overview of the I/O Subsystem 9-2
Figure 10-1. Conversational Transfer Mode 10-10
Figure 10-2. Conversational-Mode Interrupt Characters—Default Values 10-13
Figure 10-3. Changing Conversational-Mode Interrupt Characters 10-14
Figure 10-4. Page Transfer Mode 10-19
Figure 10-5. Page-Mode Interrupt Characters—Default Values 10-20
Figure 10-6. Changing Page-Mode Interrupt Characters 10-20
Figure 10-7. Enabling BREAK 10-25
Figure 10-8. BREAK Access Established After Pressing the BREAK Key 10-28
Figure 10-9. BREAK Access Established Before Pressing the BREAK Key 10-29
Figure 11-1. Modified VFC Table 11-25
Figure 12-1. Physical Tape Records Containing Records of Unspecified Length 12-15
Figure 12-2. Physical Tape Records Containing Records of Fixed Length 12-16
Figure 12-3. Example of Buffered-Mode Operation 12-20
Figure 12-4. Example of a WRITE Error in Buffered Mode 12-86
Figure 14-1. An IOEdit Packed Line 14-6
Figure 15-1. File-Name Conventions for SIO File 15-9
Figure 15-2. The INITIALIZER Process 15-12
Figure 15-3. Nowait I/O Applied to a Single SIO File 15-40
Figure 15-4. Nowait I/O Applied to Multiple SIO Files 15-42
Figure 15-5. Requester and Server Processes in Two-Way Communication 15-47
Figure 15-6. Requester and Server Processes in One-Way Communication (pTAL Version) 15-50

Figure 15-7. Requester and Server Processes in One-Way Communication (TAL Version) 15-51
Figure 16-1. TNS and Native Process Code Spaces 16-7
Figure 16-2. Mom and Ancestor Processes 16-12
Figure 16-3. Job Ancestor Relationships 16-14
Figure 16-4. Execution Priority Example 16-17
Figure 16-5. Process States 16-18
Figure 16-6. Runnable Processes 16-19
Figure 16-7. Running a Process at a High PIN or a Low PIN 16-30
Figure 17-1. The User Data Segment 17-2
Figure 17-2. The Data Stack 17-5
Figure 17-3. Data Segments for TNS/R Native Processes 17-8
Figure 17-4. Native Main Stack 17-12
Figure 17-5. Relative Location of Selectable Segments 17-18
Figure 17-6. Relative Location of Flat Segments 17-20
Figure 17-7. Referencing a Selectable Segment 17-31
Figure 17-8. Referencing Flat Segments 17-33
Figure 17-9. Defining a Memory Pool 17-44
Figure 17-10. Getting Space in a Memory Pool 17-46
Figure 18-1. Time Management Procedures 18-5
Figure 18-2. Elapsed Time and Process Time 18-15
Figure 19-1. Format-Directed Formatting 19-3
Figure 19-2. Formatting Input 19-5
Figure 19-3. Contents of a Data Descriptor 19-7
Figure 19-4. Formatting Numbers and Text 19-10
Figure 19-5. Buffer Control 19-12
Figure 19-6. Formatting Literals 19-14
Figure 19-7. Tabulating Data 19-17
Figure 19-8. List-Directed Formatting 19-26
Figure 21-1. Functions of a Requester Process 21-2
Figure 21-2. Server Selection by Function 21-3
Figure 21-3. Selecting From Functionally Identical Servers 21-4
Figure 21-4. The Requester in the Example Application 21-11
Figure 21-5. Relationship Between Major Procedures in the Requester Program 21-14
Figure 22-1. Single-Threaded and Multithreaded Servers 22-2
Figure 22-2. Server Processes in the Example Application 22-10
Figure 22-3. Relationship Between Major Procedures in the Part-Query Server 22-11

Figure 22-4. Relationship Between Major Procedures in the Process-Order Server 22-28
Figure 22-5. Relationship Between Major Procedures in the Order-Query Server 22-48
Figure 23-1. Relationships Between TACL Processes and $CMON 23-2
Figure 23-2. $CMON With Operator Control Process 23-28
Figure 25-1. Trap Handler Data Stack When Trap Occurs 25-12
Figure 25-2. Trap Handler Data Stack After Storage Allocation 25-13
Figure 26-1. Binary Semaphore 26-2
Figure 27-1. Active Backup Program Structure 27-8
Figure 27-2. Updating State Information 27-12
Figure 27-3. Control State Example 27-14
Figure 27-4. Using Switch Statement to Determine the Continuation Point 27-15
Figure 27-5. Primary Process Functional Flow 27-33
Figure 27-6. Backup Process Functional Flow 27-34

Tables

Table 10-1. Terminal CONTROL Operations 10-3
Table 10-2. Summary of Terminal SETMODE Functions 10-3
Table 11-1. Printer CONTROL Operations 11-3
Table 11-2. Printer SETMODE Functions 11-3
Table 11-3. Common PCL Escape Sequences 11-8
Table 11-4. Default VFC Table 11-22
Table 11-5. Modified VFC Table 11-24
Table 12-1. Magnetic Tape CONTROL Operations 12-2
Table 12-2. Magnetic Tape SETMODE Functions 12-3
Table 12-3. SETMODE 66 parameter-1 Settings for Tape Density 12-72
Table 12-4. Commonly Encountered Tape Errors 12-87
Table 14-1. Advantages of IOEdit Over SIO 14-2
Table 14-2. Advantages of SIO Over IOEdit 14-4
Table 14-3. Functions of the OPENEDIT_ Procedure 14-8
Table 14-4. Record Numbers 14-11
Table 14-5. Effects of the BACKSPACEEDIT Procedure 14-17
Table 16-1. Priority Values for System and User Processes 16-16
Table 25-1. Summary of Trap Conditions 25-10
Table 25-2. Map of Signals to Traps 25-22
Table 25-3. Signals Functions That Conform to the POSIX.1 Standard 25-24
Table 25-4. HP Signals Extensions to the POSIX.1 Standard 25-27
Table 26-1. Process Synchronization 26-3
Table 26-2. Binary Semaphore Procedure Calls 26-4
Table 27-1. Differences Between C Active Backup and TAL Passive Backup 27-30

What’s New in This Manual

Manual Information

Abstract

This guide describes how to access operating-system services from an application program using Guardian™ procedure calls.

Product Version

N/A

Supported Release Version Updates

This manual supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.25 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
421922-014	August 2012

Document History

Part Number	Product Version	Published
421922-009	N/A	August 2010
421922-010	N/A	January 2011
421922-011	N/A	August 2011
421922-013	N/A	February 2012
421922-014	N/A	August 2012

New and Changed Information

Changes to the 421922-014 manual

Updated the following for the J06.14 and H06.25 RVUs:

- Corrected the programming example in Mixed Data Model Programming on page A-1.
- Updated TSMSGIP process information for RVUs beginning with J06.12 and H06.23 in Controlling the IPU Affinity of Processes on page 16-64.
- Updated system clock information throughout Managing Time on page 18-1.
- Updated binary semaphore information throughout Synchronizing Processes on page 26-1.

Changes to the 421922-013 manual:

Added an appendix to describe Mixed Data Model Programming.

Changes to the 421922-011 manual:

- Rephrased the words Processor and Processor Module to CPU and to IPU at few instances throughout the document.
- Updated the first sentence in Spreading the Workload Among Multiple CPUs.
- Updated the following in the glossary:
    - Rephrased Processor Module term used in Process Environment definition in the Glossary to IPU.
    - Reworded Processor to CPU and replaced its definition.
    - Added the term IPU and defined it in the Glossary.
- Updated and added the following information in chapter 16:
    - Updated the first sentence in Process Priority section.
    - Updated the sentence The example in Figure 16-4 shows shows how CPU time is divided among three processes executing in the same IPU.
    - Added Controlling the IPU Affinity of Processes to chapter 16.
    - Added Control the placement of processes onto IPUs (IPUAFFINITY_GET_, IPUAFFINITY_SET_, and IPUAFFINITY_CONTROL_). to the existing list.
    - Rephrased the paragraphs after Figure 16-16 on page 16-19.
    - Added Controlling the IPU Affinity of Processes section at the end of the chapter.

Changes to the 421922-010 manual:

- Updated the following figures for image quality reasons:
    - Figure 3-3, Avoiding the Two-Process Deadlock, on page 3-9.
    - Figure 5-6, Example of Alternate-Key File for Use With an Entry-Sequenced File, on page 5-72.
    - Figure 12-3, Example of Buffered-Mode Operation, on page 12-20.
    - Figure 15-2, The INITIALIZER Process, on page 15-12.
    - Figure 15-5, Requester and Server Processes in Two-Way Communication, on page 15-47.
    - Figure 16-7, Running a Process at a High PIN or a Low PIN, on page 16-30.
    - Figure 17-6, Relative Location of Flat Segments, on page 17-20.

  - Figure 27-1, Active Backup Program Structure, on page 27-8.
  - Figure 27-5, Primary Process Functional Flow, on page 27-33.
  - Figure 27-6, Backup Process Functional Flow, on page 27-34.
- Added the following tables:
  - Table 11-4, Default VFC Table, on page 11-22.
  - Table 11-5, Modified VFC Table, on page 11-24.

Changes to the 421922-009 Manual

- Updated the description of Using DEFINEs on page 7-1.
- Updated the sequence in the Reading the Startup Sequence Without INITIALIZER section on page 8-17 and updated the example on page 8-18.
- Updated the example in the Specifying a Process Name section on page 16-24.
- Updated the description about Clock Averaging and System Time on page 18-1.

Changes to the H06.14/J06.03 Manual

- References to Release Version Updates (RVUs) throughout this manual have been updated to include references to J-series RVUs, where appropriate.

Changes to the 421922-004 Manual

- Unless otherwise indicated in the text, discussions of native mode behavior, processes, and so forth apply to both the TNS/R code that runs on G-series systems and to the TNS/E code that runs on H-series systems.
- Section 1, Introduction to Guardian Programming discusses Similarities and Differences Between H-series RVUs on the TNS/E Platform and G-Series RVUs on the TNS/R Platform.
- Section 16, Creating and Managing Processes, includes new information about Process Organization and Data Spaces for Native Processes for TNS/E native processes.
- Section 17, Managing Memory includes information about managing memory in TNS/E systems:
  - Managing the TNS User Data Segment
  - Managing the Native User Data Areas
  - How the Main and Priv Stacks Are Used
  - Changing the Maximum Size of the Heap
  - Changing the Maximum Size of the Main Stack

  - Overview of Flat Segments
  - Allocating a Selectable Segment
  - Allocating a Flat Segment
  - Using Memory Pools
- Section 25, Debugging, Trap Handling, and Signal Handling includes information about using the H-series debuggers.
- Section 28, Using Floating-Point Formats adds that the IEEE floating-point format is the default floating-point format on H-series NonStop operating system and later product versions.

About This Manual

This guide describes how to call Guardian procedures from your application program to obtain services from the operating system. Except where noted, the Guardian programming environment is assumed.

This guide is intended for intermediate and advanced application programmers who are not familiar with the operating system. You should already be familiar with the high-level programming language in which you are implementing your application. To get the most out of this guide, you should also have some knowledge of the Transaction Application Language (TAL).

Contents

This guide provides details of how to access services that are accessible through Guardian procedure calls to the operating system or file system. Many complete programming examples illustrate the use of these features. These examples start off simple in the early sections and become increasingly complex as the guide progresses. Toward the end of the guide is a complete example of a requester/server application.

This guide is organized as follows:

- Section 1, Introduction to Guardian Programming provides an overview of the procedure-call interface to the operating system, including an introduction to the requester/server application model.
- Section 2, Using the File System through 15 describe how to use the file system. This includes the use of file-system elements to communicate with disk files and device files, as well as communication with other processes. In addition, access to EDIT files using the sequential input/output (SIO) and IOEdit procedures is also discussed.
- Section 16, Creating and Managing Processes discusses process management, including how to create and delete processes, suspend and activate processes, and retrieve process information.
- Section 17, Managing Memory shows how to manage memory, including how to manage the user data segment, how to access data in extended data segments, and how to use memory pools.
- Section 18, Managing Time shows how to manage the time-related features of the operating system.
- Section 19, Formatting and Manipulating Character Data shows how to use procedure calls to format and manipulate character data, including how to work with multibyte character sets.
- Section 20, Interfacing With the ERROR Program shows how to interface with the ERROR program.

- Sections Section 21, Writing a Requester Program and Section 22, Writing a Server Program provide working examples of requester and server programs.
- Sections Section 23, Writing a Command-Interpreter Monitor ($CMON) and Section 24, Writing a Terminal Simulator show how to write a Tandem Advanced Command Language (TACL) monitor program ($CMON) and how to write a terminal simulator.
- Section Section 25, Debugging, Trap Handling, and Signal Handling provides debugging, trap-handling, and signal-handling information.
- Section 26, Synchronizing Processes shows how to use the binary semaphore procedure calls to synchronize processes for the purpose of accessing a shared resource.
- Section 27, Fault-Tolerant Programming in C explains how to use process pairs to write fault-tolerant programs in the C programming language.
- Section 28, Using Floating-Point Formats explains how to use the floating-point formats.

Related Reading

While using this guide, you will also need to refer to some of the manuals described here. Figure i shows the relationship between this guide and the manuals you are most likely to need to refer to. The following paragraphs provide a complete list. The following manuals appear in D-series, G--series, and H-series collections in the NonStop Technical Library, but each manual may not be supported in all collections.

Manuals Containing Procedure-Call Information

The following manuals contain information related to Guardian procedure calls:

- The *Guardian Procedure Calls Reference Manual* contains a detailed description of the syntax for most of the Guardian procedure calls.
- The *Guardian Programming Reference Summary* provides a summary of procedure call syntax, interprocess messages, error codes, and other material in a quick-reference format.
- The *Guardian Procedure Errors and Messages Manual* describes error codes for Guardian procedures, error lists, interprocess messages, and trap numbers.

Manual Containing OSS Programming Information

The *Open System Services Programmer's Guide* contains programming information related to OSS system and library function calls, as well as information on using Guardian procedure calls from the OSS environment.

Manual Containing Application Availability Information

The *Availability Guide for Application Design* provides an overview of application availability options available to software designers and developers.

Manuals Containing DSM Programming Information

The following manuals contain information for writing Distributed Systems Management (DSM) applications:

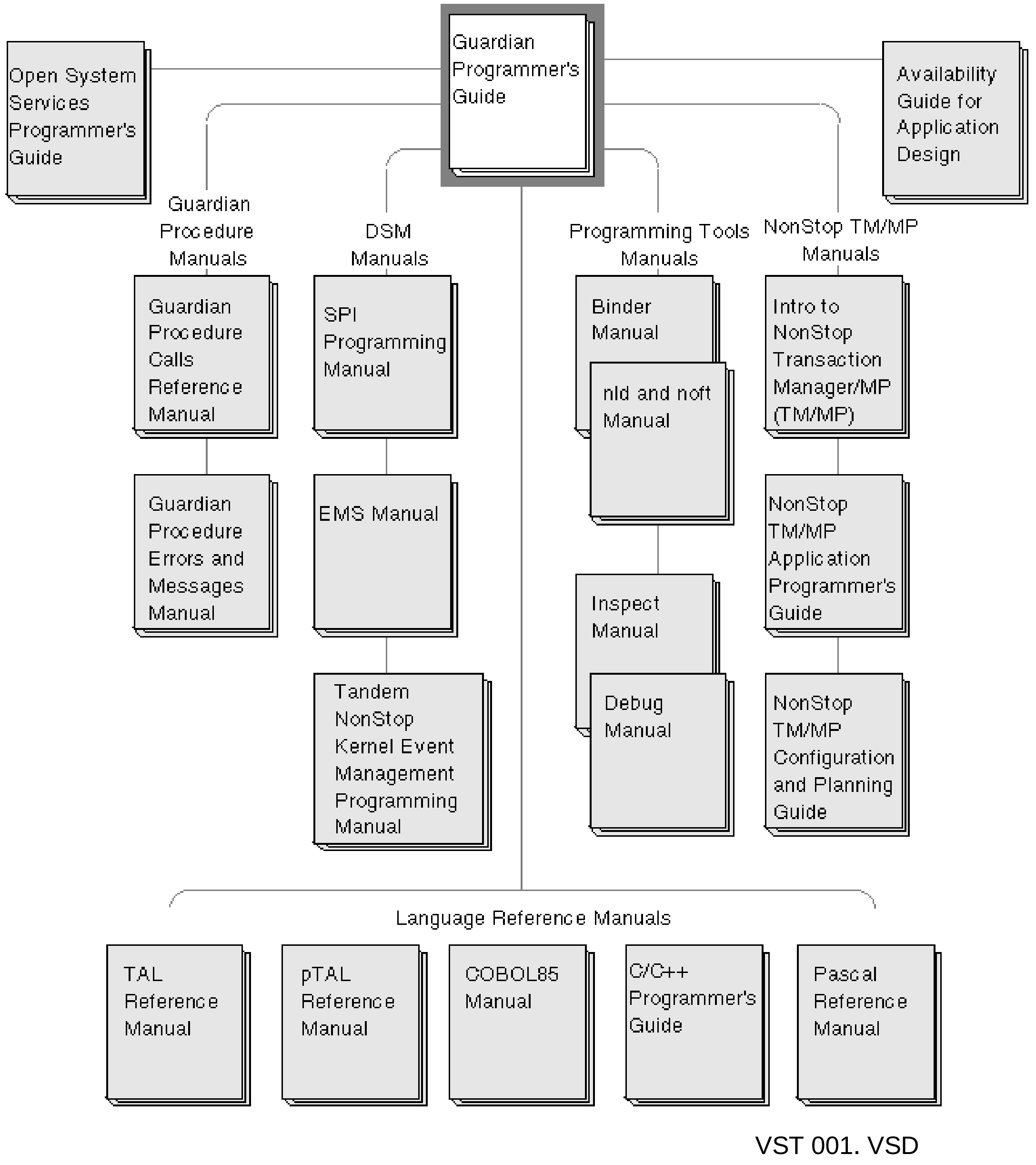
- The *SPI Programming Manual* describes the Subsystem Programmatic Interface and how to use it in a DSM application.
- The *EMS Manual* describes the Event Management Service, which allows an application to collect, process, distribute, and generate event messages.
- The *Tandem NonStop Kernel Event Management Programming Manual* describes NonStop operating system event messages.

Manuals About the Command Interface

The following manuals contain information related to the command interface:

- The *Guardian User's Guide* provides information about the Guardian command interface to the operating system. Useful primarily for inexperienced end users, this guide tells how to use the most commonly used commands and utilities.
- The *TACL Reference Manual* provides a reference for all TACL commands and built-in functions, arranged alphabetically.

Figure i. Related Manuals


Guardian Programmer's Guide
Open System Services Programmer's Guide
Availability Guide for Application Design
Guardian Procedure Manuals
DSM Manuals
Programming Tools Manuals
NonStop TM/MP Manuals
Guardian Procedure Calls Reference Manual
SPI Programming Manual
Binder Manual
nld and noft Manual
Intro to NonStop Transaction Manager/MP (TM/MP)
Guardian Procedure Errors and Messages Manual
EMS Manual
NonStop TM/MP Application Programmer's Guide
Inspect Manual
Debug Manual
Tandem NonStop Kernel Event Management Programming Manual
NonStop TM/MP Configuration and Planning Guide
Language Reference Manuals
TAL Reference Manual
pTAL Reference Manual
COBOL85 Manual
C/C++ Programmer's Guide
Pascal Reference Manual
VST 001. VSD

Manuals Describing Programming Languages and Tools

The following manuals contain reference information for writing programs in high-level languages.

- *TAL Programmer's Guide*
- *TAL Reference Manual*
- *pTAL Reference Manual*
- *pTAL Conversion Guide*
- *COBOL85 Manual* Note: This manual is replaced in the H-series by the *HP COBOL Manual for TNS and TNS/R Programs* and the *HP COBOL Manual for TNS/E Programs*
- *C/C++ Programmer's Guide*
- *Pascal Reference Manual*
- *FORTRAN Reference Manual*
- *TACL Reference Manual*

The following manuals describe tools used in program development:

- The *nld and noft Manual* describes `nld`, the Native Mode Linker, and `noft`, the Native Object File Tool. These utilities replace Binder for native mode. Note: This manual is replaced in the H-series by the *nld Manual* and the *noft Manual*
- The *Binder Manual* describes Binder, an interactive linker that allows you to examine, modify, and combine object files and to generate load maps and cross-reference listings.
- The *Inspect Manual* describes the Inspect program, an interactive debugger. You can use the Inspect program as either a source-level symbolic debugger or a machine-level debugger. Note: This manual is replaced in the H-series by the *Native Inspect Manual.*
- The *Debug Manual* describes Debug, a machine-level interactive debugger.

Manuals Containing Native Migration Information

The following manuals contain information introducing new NonStop operating system features and steps required to migrate programs to these new platforms..

- The *TNS/R Native Application Migration Guide* introduces the TNS/R native compilers and utilities and the TNS/R execution environment.
- The *H-Series Application Migration Guide* introduces the TNS/E native compilers and utilities and the TNS/E execution environment.

Database-Related Manuals

The following manuals contain programming material for writing programs that access either Enscribe data files or a NonStop SQL/MP database:

- *Enscribe Programmer’s Guide*
- *NonStop SQL/MP Reference Manual*
- *NonStop SQL Programming Manual for TAL*
- *NonStop SQL/MP Programming Manual for COBOL85*
- *NonStop SQL/MP Programming Manual for C*
- *NonStop SQL Programming Manual for Pascal*
- *Introduction to NonStop Transaction Manager/MP (TM/MP)*
- *NonStop TM/MP Application Programmer’s Guide*
- *NonStop TM/MP Reference Manual*
- *NonStop TM/MP Configuration and Planning Guide*

Device-Related Manuals

The following manuals contain information about programmer utilities that you can use with disk files and devices:

- *File Utility Program (FUP) Reference Manual*
- The spooler manuals
- *Guardian Disk and Tape Utilities Reference Manual*

The following manuals contain programming information for accessing terminals and printers:

- *Asynchronous Terminal and Printer Processes Programming Manual*

System Generation and Configuration Manuals

The following manuals contain D-series system generation and configuration information. Of interest to the programmer here is how to configure devices on your system:

- *System Generation Manual*
- *System Generation Manual for Terminals and Printers*
- *System Generation Manual for Disk and Tape Devices*
- *Dynamic System Configuration (DSC) Manual*
- *Introduction to Tandem NonStop Systems*

The following manuals contain configuration information for G-series releases.

- *System Generation Manual for G-Series Releases*
- *SCF Reference Manual for the Kernel Subsystem*
- *SCF Reference Manual for G-Series Releases*
- *SCF Reference Manual for the Storage Subsystem*
- *Asynchronous Terminal and Printer Processes Programming Manual*
- *SCF Reference Manual for Asynchronous Terminals and Printer Processes*
- *Himalaya S-Series Planning and Configuration Guide*
- *TCP/IP Configuration and Management Manual*
- *Introduction to Tandem Himalaya S-Series Servers*
- *LAN Configuration and Management Manual*
- *WAN Subsystem Configuration and Management Manual*

System and Internal Design Manuals

The system description manual for your system provides in-depth background information about system architecture.

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`

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] 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 [ , 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 % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

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

Notation for Management Programming Interfaces

UPPERCASE LETTERS. Uppercase letters indicate names from definition files; enter these names exactly as shown. For example:

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

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

```
token-type
```

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

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

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

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

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.

Legal Notices

© Copyright 2012 Hewlett-Packard Development Company L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

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

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Itanium, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

> OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
>
> OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.
>
> © 1990, 1991, 1992, 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:
>
> © 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation.
>
> This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.
>
> Printed in the US

HP Encourages Your Comments

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

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

1

Introduction to Guardian Programming

Writing an application program requires an understanding of the environment and services provided by the operating system. This guide describes how to use Guardian procedures in your application program to obtain services from the HP NonStop operating system and the file system (that is, from the operating system).

This section introduces some of the key topics covered in this guide and provides references to other sections that contain more detailed information. This section provides an overview of:

- The role of the operating system in providing fault tolerance
- The operating-system services available to the application programmer
- The requester/server application design that much of this guide supports
- How to call Guardian procedures from an application program
- The program execution modes that are available on TNS/R systems
- How to use the parameter declarations files
- How to synchronize processes

Providing Fault Tolerance

The basic design philosophy of fault tolerance is that no single module failure will stop or contaminate the operating system. This capability is called fault-tolerant operation. Redundant hardware, backup power supplies, alternate data paths and bus paths, redundant controllers, and mirrored disks all contribute to the fault tolerance of the operating system. The *Introduction to Tandem NonStop Systems* describes these features.

There is more to fault tolerance than hardware. Fault tolerance requires that all programs, the operating system as well as individual application programs, contribute to the reliability and recoverability of a process if a failure occurs. Therefore, fault tolerance should be considered from both the hardware and software perspectives.

For information about options for achieving fault tolerance in software applications, refer to the *Availability Guide for Application Design*.

Application-Level Fault Tolerance

There are several ways in which an application can be designed to withstand operating-system failures. Three such methods are introduced below:

- Transaction protection using the NonStop Transaction Manager/MP (TM/MP)
- Process pairs
- Persistent processes

Any combination of these techniques could be appropriate for providing fault tolerance, depending on the needs of the application.

The Transaction Approach to Fault Tolerance

Using TM/MP software, fault tolerance is achieved by grouping operations into transactions. At the start or end of a transaction, your data is always in a consistent state. If any kind of failure occurs during the transaction, then the transaction is “backed out” by rolling back the data to the known consistent state at the start of the transaction. The transaction can then be restarted using consistent data. See the *Introduction to NonStop Transaction Manager/MP (TM/MP)* for details.

Process Pairs

You use function calls (in C programs) or Guardian procedure calls to provide fault tolerance in your application by means of process pairs: a primary process performs the application, while a secondary (backup) process in another CPU remains ready to take over if the primary fails. The primary process uses checkpoints to copy selected parts of its environment to the backup. Using this checkpointed information, the backup process is able to take over from the primary without interrupting service to the user of the application.

The process-pair technique can be used to protect data that cannot be considered part of a transaction and therefore cannot be protected by the transaction mechanism; for example, information that remains in memory and does not get written to disk.

Writing fault tolerant programs using the C language is described in Section 27, Fault-Tolerant Programming in C.

Persistent Processes

Processes that only supply services to other processes but otherwise maintain no data of their own need only to continue to execute. For such processes, it might be appropriate simply to ensure that the process gets restarted whenever it stops. A monitor process that periodically checks the process status can restart such a process. Processes monitored in this way are sometimes called “persistent processes.”

Mirrored Disks

One effective protection against loss of data is the use of mirrored disk volumes. Mirrored disk volumes maintain copies of data on two physically independent disk drives that are accessed as a single device and managed by the same I/O process. All data written to one disk is also written to the other disk. All data read from one disk could be read from the other, because the data is identical. A mirrored volume protects data against single-disk failures: if one disk drive fails, the other remains operational. The odds against both disk drives failing at the same time are high when you always have a disk drive repaired or replaced promptly if it fails.

After a disk is replaced or a drive is repaired, all data is copied back onto it when the operator issues a Peripheral Utility Program (PUP) REVIVE command (on D-series releases) or Subsystem Control Facility (SCF) REVIVE command (on G-series releases). Processing continues while the revive operation takes place. Mirrored operation resumes as the transfer of data begins.

Multiple Copies of the Operating System

Each CPU has its own copy of the operating system. If a failure of one processing module should occur, then each other processing module has its own operating system copy to allow it to continue. Moreover, a failure in the operating system is confined to the CPU in which the failure occurs, without affecting the other processing modules.

System Integrity

Concurrent with application program execution, the operating system continually checks the integrity of the system. Each CPU transmits “I’m alive” messages to all other CPUs at a predefined interval (approximately once per second). Following this transmission, each CPU checks for the receipt of an “I’m alive” message from each of the other CPUs.

In addition to sending “I’m alive” messages to other CPUs, each CPU periodically tests its ability to send and receive messages by sending messages to itself on both buses. Unless it regularly receives messages from itself on at least one bus, it halts to ensure that it will not interfere with the correct operation of other CPUs.

If the operating system in one process module fails to receive “I’m alive” messages from another process module then it responds as follows. The operating system groups the CPUs that are able to send messages to themselves and others. CPUs show that they are operational by joining the group; any modules that do not join the group within a short period of time are declared nonoperational.

System Services

You can access the services supported by the operating system in two ways:

- By making calls from an application program to Guardian procedures

- By interacting with the Tandem Advanced Command Language (TACL) command interpreter

This guide describes how to use Guardian procedures. For information about entering commands at the command-interpreter prompt, see the *TACL Reference Manual*.

The following paragraphs describe the system services available to the application or system programmer through Guardian procedures and provide an overview of how these services might be used when writing an application.

The File System

The file system provides access to data and devices. Specifically, the file system provides the following services to the application or system programmer:

- File identification through file names
- Control over concurrent access to files
- Waited and nowait I/O
- Access to structured and unstructured disk files through the Enscribe database record manager
- Communication between processes (interprocess communication)
- The ability to perform file-name substitution or pass values to a process using DEFINEs
- Access to devices

The following paragraphs introduce each of these services and two additional sets of routines that are related to the file system but not part of it: the SIO (sequential input/output) routines and the IOEdit routines.

Files and File Names

The file system provides a set of Guardian procedures that you can use not only to access files on disk but also to access a wide variety of other entities, including terminals, printers, tapes, and processes; in other words, anything your program can do I/O to. It is possible to do this because the file system treats all these entities as files. This way, the file system is able to mask as far as practical the differences between devices but give access to file-type-specific features where needed.

A file name is not necessarily the name of a disk file. A file name is a character string presented to a Guardian procedure (such as the FILE_OPEN_ procedure) in order to open a connection through the file system. The file name identifies an object to read or write, such as a disk file, a terminal, a printer, or a process.

Section 2, Using the File System, describes files and file names in detail and provides information on how to perform common tasks on files, such as opening, closing, reading, and writing.

Operations that you can perform on file names are described in Section 13, Manipulating File Names. For example, you can scan a string of characters to see whether it contains a valid file name, or you can modify portions of a file name.

Concurrent File Access

Because the operating system provides a multiprocessing environment, it is possible that more than one process may try to access the same data concurrently. The operating system therefore provides services that each process can use when accessing data and devices to protect them from corruption by other processes. You can apply this protection at the file or record level.

Section 3, Coordinating Concurrent File Access, provides details of file-level locking and concurrency control for all types of files. Section 5, Communicating With Disk Files, provides information about locking at the record level for disk files.

Waited and Nowait I/O

Having initiated an I/O operation, a process normally waits for the I/O operation to finish before it continues. This operation is known as waited I/O.

In nowait I/O, a process continues processing after initiating the I/O. The process and the I/O then proceed in parallel. This feature of nowait I/O can be used, for example, in a process that prompts several terminals for input. Such a process can have several I/O operations outstanding at once, and it can then respond to the first terminal that responds.

In addition to allowing the application to proceed in parallel with an I/O operation, nowait I/O can also be used to specify a time limit for an I/O operation. For example, an application can prompt a user to log on to the application, then stop itself if the user does not respond within a given time.

Section 4, Using Nowait Input/Output, provides details of waited and nowait I/O.

Disk-File Access

Disk files are either NonStop SQL files or Enscribe files. For information on how to access NonStop SQL files, see the NonStop SQL manuals. This guide discusses Enscribe files.

Using the Enscribe database record manager, you can work with key-sequenced files, entry-sequenced files, and relative files, as well as with unstructured files.

Section 5, Communicating With Disk Files, describes access to Enscribe disk data files using Guardian procedure calls.

In addition to accessing Enscribe files directly through the Guardian procedures described in this guide, you can also access disk files using the TM/MP software; see the *NonStop TM/MP Application Programmer’s Guide* for details.

Interprocess Communication

The message system allows processes to pass messages to each other. This subsystem not only provides critical links between various process modules within the operating system itself but also provides the mechanism for user processes to send messages to each other and for user processes to receive messages from the operating system.

Interprocess communication is done using request-response interactions. The term “message” is used to refer to the request part of the communication. The response is usually called a “reply.” The reply can contain data, or it might contain no useful data other than an acknowledgment that the request has been accepted. However, there is always a reply of some kind.

To send messages to another process, you first establish a connection to that process by opening the process and then write to the file number returned by the open. Because a process might not know in advance which processes will send messages to it and in which order, all messages to a process arrive using a single file-system connection that is established when the process opens a file using the special name of $RECEIVE.

Section 6, Communicating With Processes, provides details on how to pass messages between user processes and receive messages from the operating system.

DEFINEs

A DEFINE is a named set of attributes and values that you use to pass information to a process before running the process. For example, you can pass labeled-tape attributes to a process using a DEFINE; this way you can make the process access a different labeled-tape file and for a different purpose each time you run it.

Section 7, Using DEFINEs, describes how to program with DEFINEs.

Devices

As already mentioned, terminals, printers, magnetic tape devices, and data communications lines are treated as files. Section 9, Communicating With Devices, provides an overview of common information about the programmatic interface to all devices.

For terminals, an application process can perform read and write operations in a way similar to reading from and writing to other files. The application can also control the operation of a terminal; for example, the program can control the action taken when the user presses the BREAK key and whether input typed by the user is displayed on the terminal screen. You can write an application program that simulates a terminal. Section 10, Communicating With Terminals, describes terminal I/O operations. Section 24, Writing a Terminal Simulator, provides information on how to write a terminal-simulation process.

For printers, the operating system allows the application not only to write data to the printer file but also to provide control operations such as advancing the paper to the top of the page or changing the character font of printers that have that capability. The printer control language (PCL) provides application control capability. Section 11, Communicating With Printers, provides details.

For magnetic tape, an application program can perform read and write operations as well as control operations such as rewinding the tape. The operating system supports both labeled and unlabeled tape. Section 12, Communicating With Magnetic Tape, describes the programmatic interface.

Sequential Input/Output (SIO) Routines

The sequential input/output (SIO) routines provide a higher-level interface than the file system. They are useful for reading or writing text streams to or from a terminal, a printer, or a disk file in EDIT format. For example, you might use SIO for command input or listing files.

A higher-level interface like the one provided by SIO is necessary for accessing files in EDIT format, because these files have a data structure in addition to what the file system understands.

The programmatic interface to the SIO routines is made up of a set of Guardian procedure calls and is described in Section 15, Using the Sequential Input/Output Procedures.

The IOEdit Routines

The IOEdit routines provide an alternative to SIO for accessing files in EDIT format. Like SIO, IOEdit provides a higher-level interface than the file system.

Section 14, Using the IOEdit Procedures, describes the programmatic interface to IOEdit.

The Startup Sequence

When a new process starts, a sequence of messages usually provides that process with some information about the process’s environment: specifically, some user-specified file names used by the process and other user-specified information in the form of ASSIGNs or PARAMs. This startup sequence is usually observed whenever one process creates another; the creating process sends the information to the new process, but it is up to the new process to decide what to do with the information. The TACL process is an example of a process that always sends this information to the processes it creates.

Section 8, Communicating With a TACL Process, provides details of the startup sequence between the TACL process and a process it creates. For information about how you can construct and send the startup message sequence from your application, see Section 16, Creating and Managing Processes.

Process Management

The Guardian procedures provide you with the ability to create and manage processes, including the ability to allocate process resources such as the CPU in which the process will run.

One of the distinguishing features about the operating system is the ability of the system to withstand failures without stopping the application. The Tandem Pathway and TM/MP products provide functions that make some types of applications tolerate system failures; the Pathway and TM/MP manuals provide details. The operating system supports the concept of process pairs for withstanding the failure of the CPU in which the application process is running.

A process pair is two executions of the same program, coordinated so that one process acts as backup to the other process. Each process runs in a different CPU. Logic within the program determines which process is the primary and which is the backup. If there is a failure in the CPU in which the primary process is running, then the backup process can take over from the primary.

The reason that the backup process is able to take over from the primary is that the program is coded to pass checkpointing messages to the backup process from the primary, thus keeping the backup process continually aware of the executing state of the primary process. If the backup process receives notification that the CPU of the primary process has failed, then the backup process assumes the role of the primary process and continues executing the application. The end user of the application remains unaware of any failure.

Section 16, Creating and Managing Processes, describes how to create and manage simple processes.

Memory Management

The operating system manages virtual address space. The organization of space for user data differs for TNS/R native processes and TNS processes. (Refer to “TNS/R Program Execution Modes,” later in this section, for descriptions of TNS/R native processes and TNS processes.)

TNS/R native processes have a main stack and a priv stack. A native process also has a globals-heap segment, which contains global data and, optionally, a heap. The main stack and the heap grow automatically as needed, up to a maximum size. The maximum size of each can be specified when a process is created. The default limit for the main stack is 1 megabyte; the default limit for the heap is 32 megabytes.

TNS processes have a user data segment, which is typically 128 KB in size. The first 64 KB of the user data segment contains global data and the user data stack. The remaining 64 KB is also available for use, but TAL and pTAL programs must manage the space themselves. The Common Run-Time Environment (CRE) manages that area for programs in other TNS languages.

For both TNS/R native processes and TNS processes, if you need more space than what is available in the data areas normally provided by the system, you must use extended data segments. There are two types of extended data segments: Flat segments, which can provide up to 128 megabytes of extra storage, and selectable segments, which can provide up to 127.5 megabytes of extra storage.

Memory pools provide a simple and efficient way to manage extended data segments. Memory pools can also be used to manage data in the user data segment.

Section 17, Managing Memory, describes memory management, including how to manage the user data segment, how to manage extended data segments, and how to use memory pools.

Time Management

The operating system provides time management in the sense of timekeeping and interval timing. “Timekeeping” means keeping track of the time of day in each CPU. “Interval timing” means the ability to control when actions occur or to report on how long an activity has been in progress. For example, you can set timeout values for certain operations or find out how long a process has been executing.

Section 18, Managing Time, describes the programmatic interface to the timing features of the operating system.

Data Manipulation

The operating system provides several features that enable you to manipulate data. These features include:

- Procedures that convert numeric data between binary values and the ASCII strings that represent them.
- A formatter that formats output data and converts input data from external to internal form.
- Support for multibyte character sets, enabling applications to support character sets that require more characters than are provided by standard ASCII code.

Section 19, Formatting and Manipulating Character Data, provides information about the programmatic interface to the data-manipulation features of the operating system.

Debugging, Trap Handling, and Signal Handling

The system provides tools for debugging object code: the Inspect, Visual Inspect, Native Inspect (on TNS/E systems) and Debug debuggers. The Inspect program is a high-level symbolic debugger. Debug is a low-level debugger, providing information at the machine-code level. Visual Inspect is a NonStop operating system symbolic debugger that uses program visualization, direct manipulation, and other techniques to improve productivity in the development or production environment. Native Inspect is a symbolic command-line debugger used for debugging TNS/E native process and snapshots on HP TNS/E systems.

Certain critical error conditions occurring during process execution prevent normal process execution. They are mostly unrecoverable. In TNS processes, these errors cause traps. In native processes, these errors cause the process to receive a signal. (Refer to TNS/E Program Execution Modes and TNS/R Program Execution Modes, later in this section, for descriptions of native processes and TNS processes.) Traps and signals are handled by trap handlers and signal handlers, respectively.

When a trap occurs in a TNS process, the default action is for the process to enter Debug (or the Inspect debugger can be explicitly specified for a process). If you prefer to write your own trap handler, you can call the ARMTRAP procedure to install your handler. Your trap handler is subsequently notified of the particular trap condition.

When a native process receives a signal, the default action is for the process to abnormally terminate. If you prefer to write your own signal handler, you can call the SIGACTION_INIT procedure to install your handler. Your signal handler is subsequently executed when the process receives a signal.

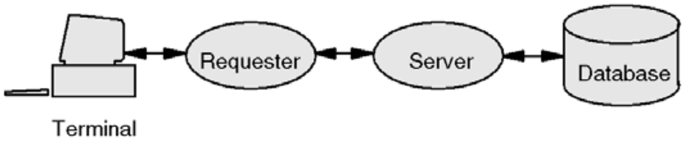
Section 25, Debugging, Trap Handling, and Signal Handling, describes the debugging, trap-handling, and signal-handling features of the operating system for Guardian processes. For debugging and signal handling in OSS processes, see the *Open System Services Programmer's Guide*.

The Requester/Server Application Model

Traditionally, application designers have placed the logic for all the functions of an application in one unified program. This program handled all aspects of the application: terminals, database, remote communication, and so on. The operating system, allows the application designer to divide the application into requester processes and server processes. These processes then communicate with each other by sending and receiving messages.

Requester processes typically represent the external user, while server processes provide most of the functional logic of the application. A typical requester/server application might have requester processes to control terminals, while server processes provide database control. Figure 1-1 shows the model in its simplest form.

Figure 1-1. A Requester/Server Application



VST002.VSD

The fact that the file system treats processes as files allows you to send user messages to them as if writing to a file. Remember that processes read interprocess messages by reading from a special input file opened with the name $RECEIVE. Each process has its own $RECEIVE input file.

Much of this guide assumes a requester/server model. Section 6, Communicating With Processes, discusses the techniques used for communication, including how to send a message to another process and how to read messages from another process. Section 21, Writing a Requester Program, and Section 22, Writing a Server Program, provide detailed examples of requester and server processes incorporating many of the features described in earlier sections.

Advantages of the Requester/Server Model

One of the advantages of the requester/server design is modularity. Once the interfaces between the processes have been defined, each process can be developed separately by its own development team. Modules developed this way are inherently easier to maintain.

Requester/server applications are a convenient design model not only because of the ease with which these applications can be developed and maintained but also because:

- You can easily add users to the application.
- You can easily add new functions.
- You can spread the load among multiple CPUs within a system.
- You can improve the performance of an application that runs on several systems in a network.

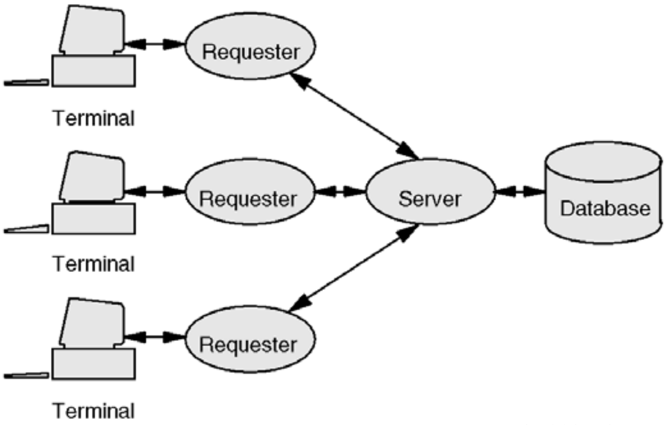
The following paragraphs discuss the above advantages of the requester/server design.

Adding Users to the Application

When you need to add new users, all you need to do is replicate the requester process. The server process is able to handle requests from several requesters. Typically, the server handles one request at a time; when the server completes a request, it waits for a request from some other requester. Alternatively, the server can be designed to process requests concurrently.

Figure 1-2 shows several requesters accessing the same server.

Figure 1-2. Multiple Users in a Requester/Server Application



VST003.VSD

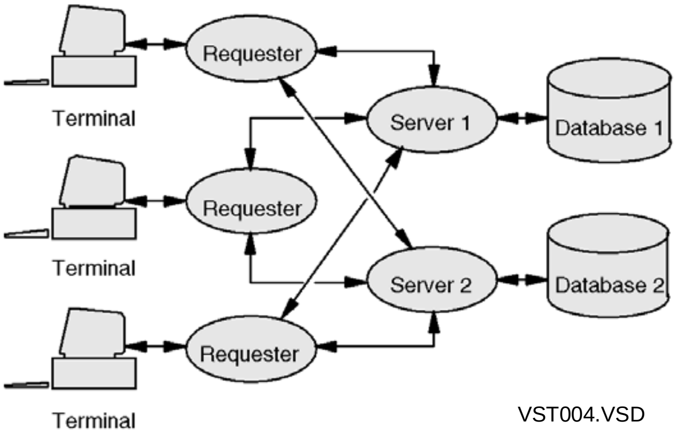
Figure 1-2 shows one terminal for each requester process. Requesters can also provide support for several terminals in each process.

Adding New Functions

If you need to add a new function to your application, you can add another server process that performs the new function. The existing server processes need no modification. Requester logic that deals with existing servers does not need changing either, but you will need some additional logic to communicate with the new server process.

Figure 1-3 shows an additional server used to maintain an additional database. When a requester makes a request against database 1, it sends an interprocess message to server 1; when a requester makes a request against database 2, it sends a message to server 2.

Figure 1-3. Multiple Functions in a Requester/Server Application


Requester
Server 1
Database 1
Terminal
Requester
Terminal
Server 2
Database 2
Requester
Terminal

VST004.VSD

Spreading the Workload Among Multiple CPUs

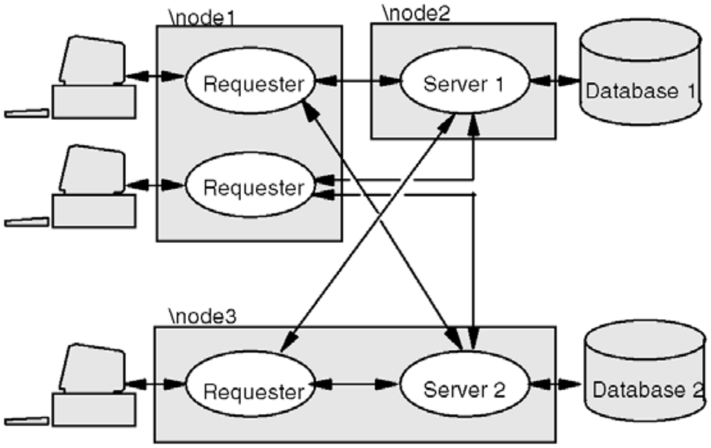
One reason why the requestor/server model works well with the operating system is the fact that you can take advantage of the NonStop multi-CPU architecture, which allows different parts of the application to run in parallel on different CPUs. Because the application is made up of several requester and server processes, these processes can be spread among the CPUs, thereby allowing parallel processing and allowing you to take maximum advantage of the processing power available in each of the system's CPUs.

Applying the Requester/Server Model in a Network Environment

Another reason why the requester/server model works well with the operating system is the fact that it can be efficiently applied to a network of systems. In a computer network, the data that a user wants to access is often controlled by some other node in the network. You can use the requester/server model to ensure that each process in an application runs on the same system as the resource that it manages. This way, the only network traffic caused by the application is interprocess messages.

Figure 1-4 illustrates the requester/server model applied to an application in a network environment. Here, each requester process runs on the same system that the user is connected to, while each server process runs on the same system as the data it manages.

Figure 1-4. Requester/Server Application in a Network Environment



VST005.VSD

Monitoring Server Processes

A monitor is a separate process that, along with other functions it might be performing, monitors and controls the execution of other processes. Because server processes must continue to run to provide needed services, one common use of a monitor is to check that each server continues running and to restart any server that stops.

A monitor is often implemented as a process pair to ensure that it survives CPU failures.

Requesters and Servers in Fault-Tolerant Applications

There are many approaches to making an application fault tolerant. There is no best method that suits all applications. The best method to use depends on the application in question.

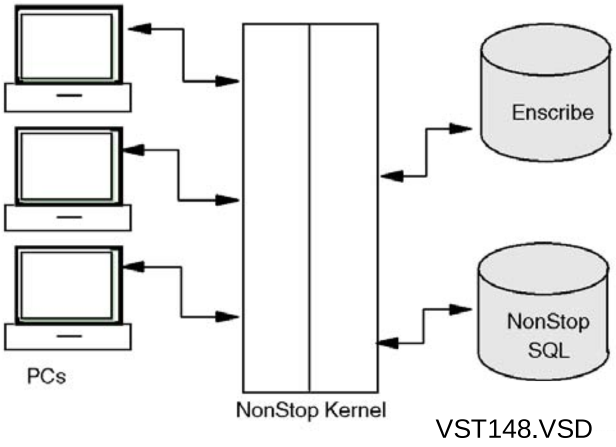
One common way of making a requester/server application fault tolerant is to run the server process as a process pair while the requesters run as simple processes. The primary server process then uses checkpoints to copy critical data to its backup process to enable a smooth transition if a failure occurs. This design makes sense in many cases, because it is the server that provides most of the functional logic of the application.

Another approach to fault tolerance is to have a fault-tolerant monitor process. If a failure occurs, then the backup process is able to restart each server process on an alternate CPU, thereby allowing the application to continue with minimal interruption.

Client/Server Application Model

The client/server model, which evolved from the requester/server model, enables the client to issue commands to the host from a GUI on a PC (see Figure 1-5 below). The client interface typically has pull-down menus, dialog boxes, color and other features that provide ease of use. In “fat” client applications, most of the processing occurs on the PC instead of the host; the client often accesses a database using a vendor-supplied product (for example, the open database connectivity). In “medium” client applications, the processing occurs on both the host and the client. In “thin” client applications, most of the processing occurs on the host. See the *Introduction to NonStop Transaction Processing* for more information about the client/server model for application development.

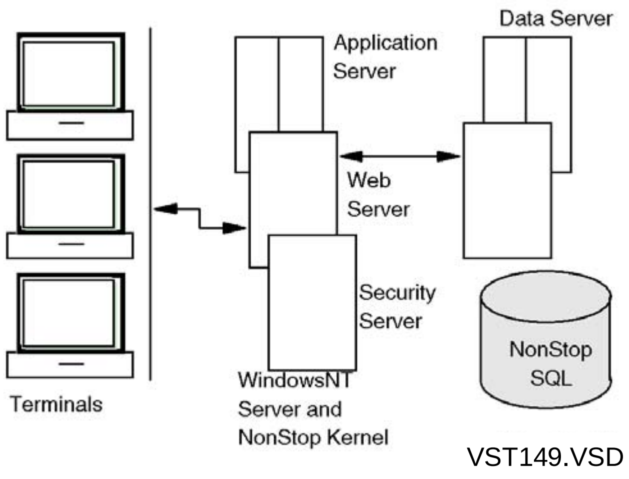
Figure 1-5. Client/Server Architecture


Enscribe
NonStop SQL
PCs
NonStop Kernel
VST148.VSD

Distributed Client/Server

Distributed applications place parts of the business logic on various servers, which can be the same or different platforms (see Figure 1-6 below). For example, the web server might provide the user interface while the two servers provide the application and database logics.

Figure 1-6. Distributed Client/Server



There are a variety of software products available for developing requester/server, client/server, and distributed client/server applications in the Guardian environment. These products include the Pathway/TS, the Remote Server Call (RSC), and the NonStop Server Object Gateway (SOG).

The Pathway/TS transaction-processing environment is designed for terminal-based requester/server applications. Pathway/TS terminal applications are written in screen COBOL, which simplifies screen definition and provides a means for invoking the servers. Server programs are written in C or COBOL.

Pathway/TS uses the run-time environment of NonStop TS/MP and NonStop Transaction Manager/MP (NonStop TM/MP) software. This means that all Pathway/TS terminal-based applications automatically acquire the NonStop fundamentals of continuous availability, data integrity, and scalability without special coding of applications. See the *Pathway/TS System Management Manual* for details.

The Remote Server Call (RSC) product enables you to develop client/server applications where UNIX and PC workstations invoke NonStop TS/MP server processes residing on the NonStop servers. Many languages, tools, and applications work with RSC, including environments that generate standard C sequences. Many other off-the-shelf tools are supported as well. RSC supports many communications protocols, including TCP/IP, NetBIOS, Asynchronous, Eicon X.25, X.25 over asynchronous, and IPX/SPX. See the Remote Server Call (RSC) manuals for details.

NonStop Server Object Gateway links popular desktop tools and critical business services using ActiveX controls. It enables any application that supports ActiveX controls to access Pathway services. SOG simplifies the development and deployment of GUI clients by shielding developers from the complexities of the transaction processing server. Pathway services appear as ActiveX objects within the client application. Client developers need no knowledge of Pathway APIs. SOG also handles communications and automatic data conversion between the client and the server using TCP/IP protocol. See the *Nonstop Server Object Gateway User's Guide* for details.

Accessing Guardian Procedures

You can access the services provided by the Guardian procedures from any supported high-level language, including C, C++, Pascal, COBOL85, and FORTRAN, as well as the Transaction Application Language (TAL) and the Portable Transaction Application Language (pTAL).

You must read the appropriate language reference manual to find out how to access Guardian procedures from the language you are using. However, you need to read this guide if your program makes calls to the Guardian procedures, regardless of the programming language you are using.

Although most of the examples in this guide are given in TAL, they have been carefully written to avoid, where possible, use of TAL features that are not normally found in other programming languages; this approach helps to make the programming examples more readable, especially if you do not normally write programs in TAL.

Calling Guardian Procedures From TAL or pTAL

Using TAL or pTAL (which compiles native RISC code), you can access Guardian procedures contained in the $SYSTEM.SYSTEM.EXTDECS0 file. This file is a source library for the external declarations for most of the procedures in the system library. However, some external declarations are not found in the EXTDECS0 file but are found in pTAL header files instead. (You cannot call these procedures from a TAL program.) Refer to the *Guardian Procedure Calls Reference Manual* for information about where external declarations are to be found for particular Guardian procedures.

Any Guardian procedure from the EXTDECS0 file that your program calls must be listed in a ?SOURCE compiler directive before the first call to that procedure appears within your program source. Typically, you include one ?SOURCE directive at the start of your program that lists all the Guardian procedures that you use; for example:

```
!global declarations

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

!procedure declarations
```

The above ?SOURCE directive copies into your program for compilation the external declarations for only the FILE_OPEN_, FILE_CLOSE_, READX, WRITEX, and WRITEREADX procedures.

Multiple versions of the external declarations file are available in case you might need to run your program on older versions of the operating system as well as the current version. Specify the version of your choice as follows:

EXTDECS0	the current operating system version
EXTDECS1	the current version minus 1
EXTDECS	the current version minus 2

For example, if the current version of the operating system is D40, EXTDECS0 relates to release D40, EXTDECS1 relates to release D30, and EXTDECS relates to release D20.

Because the version is specified by a relative value, there is no need to constantly update the level of the external declarations file used in your programs. If you specify the level as EXTDECS0, your programs will always use the current version of the external declarations file.

Calling Guardian Procedures From C or C++

HP C provides a library file known as the `cextdecs` header to help you make calls to Guardian procedures from the C and C++ languages. The `cextdecs` header contains C-coded declarations that enable most of the Guardian procedures to be called directly through C or C++ function calls. However, some declarations are not found in the `cextdecs` header, but are found in other C header files instead. Refer to the *Guardian Procedure Calls Reference Manual* for information about where declarations are to be found for a particular Guardian procedure.

Guardian procedures that return both a return value and a condition code cannot be called directly from C or C++, nor can Guardian procedures that take the name of another procedure as a parameter. For each of these calls you must:

- Supply a “jacket” procedure in a TAL module. This jacket procedure must call the desired Guardian procedure and then return the information to the caller of the jacket procedure in a way that can be handled by the C or C++ function.
- Provide a function declaration in your C or C++ program to call the jacket procedure.

The *C/C++ Programmer’s Guide* provides complete details on how to call all Guardian procedures from a C or C++ program, whether the call is direct or indirect.

For information on when and how to use calls to the Guardian procedures in your C or C++ program, you should continue to read this guide. Although most of the examples are given in TAL, the program logic is similar for both languages.

Calling Guardian Procedures From Pascal

The PEXTDECS file makes it easy to import Guardian procedures into a Pascal program. This file contains Pascal-coded definitions for Guardian procedure calls, enabling you to invoke Guardian procedures as Pascal functions. To pass parameter values to the Guardian procedure call, you simply pass the equivalent parameters to the corresponding Pascal function.

For complete instructions on how to call Guardian procedures from a Pascal program, see the *Pascal Reference Manual* .

For information on when and how to use calls to the Guardian procedures in your Pascal program, you should continue to read this guide. Although the examples are given in TAL, the program logic is similar for both languages.

Calling Guardian Procedures From COBOL85

All Guardian procedures that you can call safely from a COBOL85 program are declared in the external declarations file for COBOL85. Multiple versions of this file exist. You can specify the version of your choice from the following:

COBOLEX0	contains the current version
COBOLEX1	contains the current version minus 1
COBOLEXT	contains the current version minus 2

The COBOL85 external declarations file enables COBOL85 programmers to call Guardian procedures using ENTER TAL statements.

The *COBOL85 Manual* provides complete instructions on how to use ENTER TAL statements to call Guardian procedures from a COBOL85 program, with examples of how to map COBOL85 parameters to TAL parameters.

As for C and Pascal programmers, COBOL85 programmers need to read this guide for information about how to use the Guardian procedures.

Calling Guardian Procedures From FORTRAN

All files required by the FORTRAN programmer to access Guardian procedures are provided with the FORTRAN compiler. Therefore, to make a Guardian procedure call, you must declare the appropriate variables and then access the Guardian procedure either through a function call if the procedure returns a value or through a subroutine call if the procedure does not return a value.

For complete details on how to call Guardian procedures from FORTRAN, see the *FORTRAN Reference Manual.*

For information on how to use Guardian procedures in a FORTRAN program, you should continue to read this guide.

TNS/E Program Execution Modes

Like the G-series environment, the H-series and J-series environment support a native execution mode for TNS/E native processes. TNS/E native compilers and tools are used to generate native code that uses the process, memory, and instruction set architectures that are native to Itanium CPUs. The H-series and J-series RVUs support native versions of C, C++, and COBOL as well as a pTAL compiler. TNS/E native execution mode can be summarized as follows.

- Programs are generated by TNS/E native compilers.
- Programs use TNS/E native process and memory architecture.
- Programs consist of TNS/E native object code.
- Code consists of Itanium instructions. Itanium instructions are executed directly on Itanium CPUs.
- A TNS/E native program can contain only TNS/E native object code. Because of architectural differences between the execution modes, interpreted object code, accelerated object code, and native object code cannot be mixed in one program file.

Similarities and Differences Between H-series RVUs on the TNS/E Platform and G-Series RVUs on the TNS/R Platform

The H-series RVUs on the TNS/E platform provide a programming environment that is similar to that provided by the G-series RVUs on the TNS/R platform. These similarities ease the transition to the new platform and facilitate the migration of applications to the new environment. Similarities include:

- Binary compatibility for TNS programs. Interpreted and accelerated execution modes are supported.
- Full support for TNS development tools. You can continue to develop TNS applications on an H-series system using familiar tools.
- A native development environment that is similar to the TNS/R native development environment. The C, C++, COBOL, and pTAL languages are supported.
  - TNS/E native development tools have the same, or added, functionality as the TNS/R native development tools.
  - In most cases, the same changes are required to migrate TNS applications to TNS/E native mode as to migrate them to TNS/R native mode.
  - In most cases, no source code changes are required to migrate TNS/R native mode programs to TNS/E native mode.
- Debugging of snapshot files. You can debug TNS, TNS/R, and TNS/E snapshot files.

- Full support for native-mode cross-compilation on the PC.
- Full support for TNS/R native compilers and linkers. You can compile and link, but not execute, TNS/R native applications on an H-series system.
- Support for FORTRAN and Pascal languages. A TNS FORTRAN compiler is provided, and both FORTRAN and Pascal accelerated object files will run on the TNS/E platform, but they must be accelerated by the H-series Object Code Accelerator (OCA).

Note. All details mentioned for H-series RVUs are also applicable to J-series RVUs unless indicated otherwise.

Differences in the G-series and H-series development environments include:

- TNS/R native object files cannot be executed on a TNS/E platform; all TNS/R native programs must be recompiled using a TNS/E native compiler.
- TNS/E native object files cannot be executed on a TNS/R platform.
- The H-series OSS environment does not support TNS execution; OSS programs must be migrated to TNS/E native mode to run on an H-series system.
- TNS/E native development tools (compilers, linkers, and certain utilities) have new names, although their functionality is nearly identical.
- Different command line and system level debugging tools are provided in the H-series RVUs.
- All H-series libraries are dynamic-link libraries (DLLs); shared run-time libraries (SRLs) are not supported. The H-series RVUs provide more extensive support for DLLs than do the G-series RVUs.
- All TNS/E native code is position-independent code (PIC); unlike the TNS/R native environment, there is no distinction between PIC and non-PIC.

TNS/E Object File Format

The TNS/E native object file format differs from that of TNS/R native object files. Key differences are:

- TNS/E native object files are in 64-bit ELF format, while TNS/R native object files are in 32-bit ELF format.
- In the Guardian environment, TNS/E native object files are type 800 files; TNS/R native object files are type 700 files.
- TNS/R native object files use the Third Eye format, which mixes symbol information needed at link time with symbol information needed only for debugging at run time. TNS/E native object files use a separate DWARF symbol table for debugging information.

TNS/E native and TNS/R native object files differ in other respects as well. See the *eld Manual* and the *enoft Manual* for details on the structure of TNS/E native object files.

TNS/R Program Execution Modes

TNS/R systems can execute TNS/R native code, TNS code, and accelerated code. User processes can run in all three of these modes.

TNS/R native code is produced by a native compiler and consists entirely of RISC instructions that have been arranged to take full advantage of the RISC architecture. A program consisting of native code is called a native program. You can produce native object code by using a native compiler. Refer to the *C/C++ Programmer's Guide* and the *pTAL Programmer's Guide* for information about native compilers.

TNS code executes TNS instructions facilitated by millicode. Millicode is assembled program code, consisting of RISC instructions, that implements various TNS low-level functions. A program consisting of TNS code is called a TNS program. You produce TNS object code by using a compiler that is not native. Actual TNS CPUs do not support D40 or later versions of the operating system.

Accelerated code is produced by the Accelerator, a program that processes a TNS object file to run more efficiently on a TNS/R CPU. An accelerated object file consists of Accelerator-generated RISC instructions as well as the original TNS instructions. A program that contains accelerated code is called an accelerated program. For more information on using the Accelerator, see the *Accelerator Manual*.

The accelerated version of an object file almost always runs faster than the TNS version; the native version of an object file almost always runs faster than the accelerated version. The actual differences in execution speed between TNS, accelerated, and native versions of the same program may vary, depending on the constructs that are used in the source code.

A TNS/R native process is a process that is initiated by executing a TNS/R native program. A native process executes in the native operating environment of the TNS/R CPU.

A TNS process is a process that is initiated by executing a TNS or accelerated program. A TNS process executes in an emulated TNS operating environment.

Using Parameter Declarations Files

HP provides a set of files that contain useful literals and data structures. Many of these literals and data structures can be used in defining parameters for Guardian procedure calls. The Data Definition Language (DDL) makes these literals available from the TAL, C, COBOL85, and Pascal programming languages.

The following files are provided in the subvolume $SYSTEM.ZSYSDEFS:

- ZSYSDDL contains the DDL declarations used to generate the other ZSYS files.

- ZSYSTAL contains literals and data structure declarations for TAL programs.
- ZSYSC contains literals and data structure declarations for C programs.
- ZSYSCOB contains literals and data structure declarations for COBOL85 programs.
- ZSYSPAS contains literals and data structure declarations for Pascal programs.

To use the DDL declarations in your application, include the appropriate ZSYS file in your program. You do this using a ?SOURCE compiler directive before your program uses any of the literals or data structures listed in the ZSYS file. Like the EXTDECS files, you need to list only the sections that contain the declarations you need.

The following example for a TAL program includes the literals declared in the FILENAME^CONSTANT and FILESYSTEM^CONSTANT sections of the ZSYSTAL file:

```
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILENAME^CONSTANT,
?                                FILESYSTEM^CONSTANT)
```

Note. Many programming examples shown throughout this guide make use of the literals in the ZSYSTAL file. You can recognize them by the first four characters, which are always "ZSYS."

Synchronizing Processes

One or more processes executing concurrently may need to share a particular resource. This sharing of resources can result in conflicts and possible errors. Binary semaphores provide a way to synchronize processes so that only one process at a time can access a shared resource. While a process is using the resource, other processes can execute concurrently until they need to use the resource; they then enter a wait state. When the original process is through with the resource, it releases its hold on the resource, and a waiting process is selected to resume execution and use the resource.

Using binary semaphores, you can maximize parallelism in processes (that is, the degree to which processes execute concurrently) while ensuring that conflicts over shared resources are avoided.

Coding programs to use binary semaphores is described in Section 26, Synchronizing Processes.

2 Using the File System

This section reviews the concept of a file and describes some of the common operations that you can use on a file. This section discusses the different types of files and describes file-name syntax. It goes on to introduce some techniques for passing file names to a process before describing how to perform the following tasks:

- How to pass a file name to a process using a DEFINE or the startup sequence of messages
- How to create and open files using the FILE_CREATE[LIST]_, FILE_OPEN_, and PROCESS_CREATE_ procedures
- How to read from a file using the READ[X] and READUPDATE[X] procedures and write to a file using the WRITE[X], WRITEREAD[X], and WRITEUPDATE[X] procedures
- How to get information about files using the FILE_GETINFO[LIST][BYNAME]_ procedures
- How to handle file-system errors using the FILE_GETINFO_ procedure
- How to close files using the FILE_CLOSE_ procedure

At the end of the section, a sample program performs many of these tasks.

Many references point to more detailed information in this guide and in other manuals.

All of the capabilities of the PROCESS_CREATE_ procedure described in this section are also available through the PROCESS_LAUNCH_ procedure, although parameters are passed in a different manner to PROCESS_LAUNCH_. How to use the PROCESS_LAUNCH_ procedure is explained in Section 16, Creating and Managing Processes.

File Concepts

Recall from Section 1, Introduction to Guardian Programming, that under the operating system, the following entities are all treated as files:

- Disk files
- Devices other than disks, such as terminals, printers, and magnetic tape drives
- Processes

Each of these entities is reviewed in the following paragraphs.

Disk Files

Disk files can be SQL files or Enscribe files. You access Enscribe files using the Enscribe database record manager. You access SQL files using the NonStop SQL product. This guide discusses access to Enscribe files. For details on SQL files, see the SQL programming manuals.

Types of Enscribe Files

The Enscribe database record manager provides access to and operations on Enscribe disk files. The Enscribe software is an integral part of the operating system. It supports the following file types:

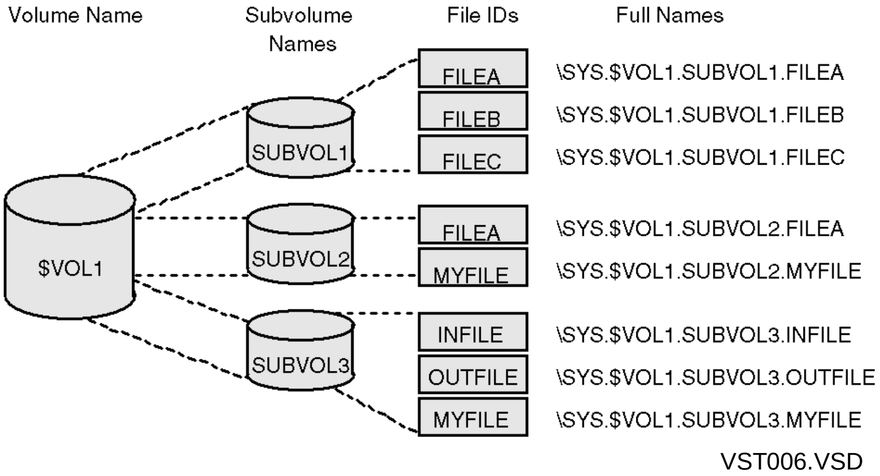
- Key-sequenced files, in which records are placed in ascending sequence based on a key field. The key field is a part of the record.
- Relative files, in which records are stored at locations relative to the beginning of the file.
- Entry-sequenced files, where records are appended to a file in the order they are written to the operating system.
- Unstructured files, in which records are defined by the application. Records are written to and read from a file using relative byte addresses within the file.

Section 5, Communicating With Disk Files, provides an overview of disk files along with programming examples of how to access and manipulate disk files. The *Enscribe Programmer's Guide* provides complete details.

Volumes, Subvolumes, and Files

The usable space of a disk (the part that can store files) is called a **volume**. For convenience, file names within the same volume that have a common middle part are treated as a logical group of files or a **subvolume**. Figure 2-1 shows how the file name reflects this organization.

Figure 2-1. Disk Files


Volume Name
Subvolume Names
File IDs
Full Names
$VOL1
SUBVOL1
SUBVOL2
SUBVOL3
FILEA
FILEB
FILEC
FILEA
MYFILE
INFILE
OUTFILE
MYFILE
\SYS.$VOL1.SUBVOL1.FILEA
\SYS.$VOL1.SUBVOL1.FILEB
\SYS.$VOL1.SUBVOL1.FILEC
\SYS.$VOL1.SUBVOL2.FILEA
\SYS.$VOL1.SUBVOL2.MYFILE
\SYS.$VOL1.SUBVOL3.INFILE
\SYS.$VOL1.SUBVOL3.OUTFILE
\SYS.$VOL1.SUBVOL3.MYFILE
VST006.VSD

Device Files

In addition to program and data files stored on disk, every terminal, printer, and magnetic tape is a file. Treating devices in this way makes device I/O as easy as accessing disk files. This approach allows disk files and devices to be handled uniformly where appropriate and allows programs to be as device-independent as possible.

What constitutes an I/O transfer with a device other than a disk depends on the characteristics of the device. On a conversational-mode terminal, for example, a transfer is one line of information; on a page-mode terminal, a transfer can be up to one page of information; on a line printer, a transfer is one line of print; on a magnetic tape unit, a transfer is one physical record on tape.

This guide discusses how to communicate with terminals, printers, and magnetic tape drives. Sections 9 through 12 provide details. For information on accessing data communications lines, refer to the appropriate data communications manual. Additional information on accessing terminals, printers, and magnetic tapes can also be found in the data communications manuals.

Process Files and $RECEIVE

The file system allows you to open and access processes as files. A process can open another process using a process file name and then send data to the process by writing to the open file.

A process can receive data from other processes by opening a file using the special file name “$RECEIVE.” Through $RECEIVE, you can read not only messages from other processes but also operating-system messages.

Section 6, Communicating With Processes, provides details on how processes communicate with each other.

File Names

Every file has at least one unique name by which the file is accessed. (Devices other than disks, but not subdevices, have two names—a regular file name and a logical device number). The file name is used by a process when gaining access to (or opening) a file. The file is named when the file is created.

The file name is unique not only on the system where the file is physically located but also within the system’s network.

Some differences exist between the form of file name you use to access a file programmatically and the form of file name you use interactively. The syntax definitions given here apply to programmatic access.

The rules for naming a file depend on whether you are naming a disk file, a device file, or a process file. The rules for each of these entities are given in the following paragraphs. Generally, the following rules apply:

- File names are made up of alphanumeric characters but can also include some of the following special characters when used as delimiters:

  \ $ # : .

- File names are not case-sensitive; $OURVOL.MYSUBVOL.MYFILE refers to the same file as $ourvol.mysubvol.myfile.

Permanent Disk-File Names

Permanent disk files are named when they are created. Once a permanent disk file is created, it remains on disk until explicitly purged. File creation is discussed later in this section.

The name of a disk file when fully qualified consists of four parts: the node name, the volume, the subvolume, and the file ID. Periods separate the parts from each other.

The syntax definition for a permanent disk file is shown below. (Temporary disk files are described later.)

Permanent disk-file name:

```
[node-name.][[volume-name.]subvolume-name.]file-id
```

Permanent disk-file names must follow these rules:

- A permanent disk-file name must be made up entirely of alphanumeric characters, except for the backslash (\) that begins the node name, the dollar sign ($) that begins the volume name, and the periods that separate the pieces of the file name. The second character of *node-name* and *volume-name* and the first character of *subvolume-name* and *file-id* must be alphabetic characters.

- Disk-file names have a maximum length of 35 characters, of which 8 characters are reserved for the *node-name* (including the backslash). The *volume-name*, *subvolume-name*, and *file-id* fields can have up to 8 characters each. (Note that the 8 characters of *volume-name* includes the dollar sign.) The following example illustrates the maximum sizes of each piece of a disk-file name:

  ```
  \nnnnnnn.$vvvvvvv.ssssssss.ffffffff
      8 + 1 +  8 + 1 + 8 +  1 + 8        = 35 characters
  ```

- A fully qualified file name contains a *node-name*, a *volume-name*, a *subvolume-name*, and a *file-id*. A partially qualified file name contains at least the *file-id* but does not contain all four parts. The *file-id* is the only mandatory part of a permanent disk-file name. The operating system provides default values for all other unspecified parts of the file name:

  2 If the *volume-name* is omitted, the default volume name is used in its place.

  2 If the *subvolume-name* and *volume-name* are both omitted, the default *volume-name* and *subvolume-name* are used.

  2 If the *node-name* is omitted, the default system is assumed.

  The default values are passed to the process from the user's =_DEFAULTS DEFINE. This DEFINE contains default values for the node name, volume name, and subvolume name. Its contents change when the user changes the current default values by issuing VOLUME, SYSTEM, and LOGON TACL commands. See Section 7, Using DEFINEs, for details of programmatic use of DEFINEs.

The following are all valid disk-file names; if \SWITCH.$DATA.MESSAGES is the default subvolume, then they all refer to the same file:

\SWITCH.$DATA.MESSAGES.ARCHIVE

$DATA.MESSAGES.ARCHIVE

MESSAGES.ARCHIVE

\SWITCH.ARCHIVE

ARCHIVE

Temporary Disk-File Names

Sometimes a file is required only as temporary work space for a program and is no longer useful once the process has terminated. Such a file is known as a **temporary file**. A temporary file must be created programmatically, and it exists only until the file is closed. The name of such a file has the following syntax:

```
Temporary disk-file name:

[node-name.][volume-name.]#temp-file-id
```

The following are valid temporary file names:

\TRANSAC.$ACCOUNT.#1234567

$ACCOUNT.#1234567

#1234567

Temporary files are created programmatically by calling the FILE_CREATE_ procedure and, usually, specifying the volume. If the volume or node name is not specified, then the default values provided by the =_DEFAULTS DEFINE are again used.

The `temp-file-id` is not specified by the program. It is returned automatically by the operating system. It always begins with a pound sign (#), followed by four to seven digits.

Device-File Names

File names also provide access to devices such as terminals, printers, magnetic tape drives, and data communications lines. A device can be accessed either by name or by logical device number.

Device names and logical device numbers are assigned either at operating-system generation time—see the *System Generation Manual* —or dynamically using the Configuration Utility Program (COUP) on D-series release systems or the Subsystem Control Facility (SCF) on G-series release systems. See the *Dynamic System Configuration (DSC) Manual* for more information about COUP and see the SCF reference manuals for more information about SCF. In either case, the assignment of device names and device numbers is the responsibility of system management, not the application programmer.

Device-file names have the following syntax:

```
Device-file name:

[node-name.] {  device-name[.qualifier]  }
             {  ldev-number               }
```

The `device-name` part of the name can be up to 8 characters long and must start with a dollar sign ($). Again, all characters must be alphanumeric, and the second character of the `device-name` part must be a letter. The `qualifier` is an optional alphanumeric string that always begins with the pound sign (#) character followed by an alphabetic character. The meaning of a qualifier depends on the device type.

We recommend using device names to identify devices. However, you can also identify a device using a logical device number that is an integer always preceded by a dollar sign. Five digits (up to 34492) are allowed in the logical device number.

Process File Names

Process file names have two forms: one for named processes and one for unnamed processes.

Process File Names for Named Processes

You can name a process at the same time you create the process either by specifying the NAME option of the RUN command or by specifying the `name-option` parameter when calling the PROCESS_CREATE_ procedure. You can accomplish the same thing with the PROCESS_LAUNCH_ procedure, although the equivalent parameters are passed as fields in a structure. Process creation is described in detail in Section 16, Creating and Managing Processes.

Assigning a name to a process hides its location in the operating system and hides whether it can reference a process pair. A process name also makes interprocess communication easier, because the name that you pass to the FILE_OPEN_ procedure is already known. On the other hand, a process that wants to communicate with an unnamed process cannot have prior knowledge of the process file name; it must establish what the process file name is at run time, then pass it to the FILE_OPEN_ call.

The syntax for file names for named processes follows:

Process file name, named process:

`[node-name.]process-name[:seq-no][.q1[.q2]]`

A named process is identified by an alphanumeric name in the `process-name` field. A `process-name` is made up of 1 to 5 alphanumeric characters beginning with a dollar sign ($). The character after the dollar sign must be a letter.

The optional sequence number (`seq-no`) enables instances of a process name to be distinguished over time. A specific process name often represents a service (for example, $S is a spooler collector), and the user does not care whether the service provider is the same instance as it was some time earlier; the user simply wants the service. The `seq-no` field is therefore often omitted. However, although failure and restart of a server is irrelevant to some requesters, it may be important to others. The operating system must therefore be able to distinguish different instances of the same named server.

The named form of the process also permits qualifiers (*q1* and *q2*) to be passed to the process. These are alphanumeric values. *q1* must start with a pound sign (#). (*q2* must not include a pound sign.) Although they are checked for correct format, these qualifiers have no meaning to the file system. Their meaning is application-dependent. When a process is opened by another process, the qualifiers are passed to the process being opened. For example, $S.#WIDE might indicate to a spooler collector process that it should direct the lines being sent to it to a printer with a line width of 132 characters; $S.#NARROW would request a printer with a line width of 80 characters.

Process File Names for Unnamed Processes

Sometimes it is necessary to refer to a process without using a process name. For example, you can identify one member of a process pair using a process file name for an unnamed process.

The syntax for file names for unnamed processes follows:

Process file name, unnamed process:

`[`*`node-name`*`.]$:`*`cpu`*`:`*`pin`*`:`*`seq-no`*

An unnamed process is identified by a combination of the CPU module number (*cpu*) and process identification number (*pin*). The process identification number (or PIN) is a unique number within a CPU.

Note that the *seq-no* field is mandatory for unnamed processes. If a process fails and some other process is created using the same CPU and PIN, the requester needs to know that the new process is not the one that it has open. Using the sequence number, the operating system is able to inform the requester that the server has failed by sending it an error condition.

Process Descriptors

A process descriptor is a limited form of a process file name. It is the form of process file name returned by Guardian procedure calls. The syntax for process descriptors follows:

Process descriptor, named process:

`node-name``.`*`process-name`*`:`*`seq-no`*

Process descriptor, unnamed process:

`node-name``.$:`*`cpu`*`:`*`pin`*`:`*`seq-no`*

Note that a process descriptor always contains a node name and a sequence number. It never contains qualifiers.

Location Independent Disk-File Names

Location independent disk-file names are supported by the NonStop Storage Management Foundation (SMF) product, which is designed to help automate system storage-management tasks. Location independent naming means that a disk file has both an external, or logical, name and an internal, or physical, name.

Normally, a disk file's name indicates the location of the file. For example, the file \SYS99.$BIGVOL.MYSUBVOL.MYFILE would designate a file located on the subvolume MYSUBVOL, on the volume $BIGVOL, on the node \SYS99. However, if this file were managed by the SMF subsystem, its location would be independent of the name, except for the name of the node.

The SMF subsystem controls the mapping of the external name of a file to the internal name. This allows the internal name, which identifies the file's physical location to the disk process, to change when a file is moved to a different location, while the external name remains the same to applications and to users. The mapping function is transparent to applications and to users.

The external name of a file managed by the SMF subsystem follows the normal syntax for a disk file name; you cannot tell that it is an SMF external name by looking at it. You can perform any normal operation on the file by using its external name.

However, there are restrictions against directly accessing an SMF file by its internal name. Also, information requests based on the internal name are disallowed unless explicitly asked for. (For example, wild-card searches either by TACL commands, such as the command FILEINFO $VOL.*.*, or by calls to the FILENAME_FIND* procedures, do not return information about files contained in the ZYT* and ZYS* subvolumes, which are reserved for SMF internal files; ZYS* and ZYT* must be specified to get information on internal files that they contain.)

For more information on the SMF product and how to use it, refer to the *NonStop Storage Management Foundation User's Guide*.

Passing File Names to Processes

There are two ways in which you can pass file names to a process:

- Using a CLASS MAP DEFINE (or other DEFINE CLASS that passes file names)
- Using the startup sequence of messages

Either of these methods allows you to use the same program to access different files without changing your program code.

These concepts are introduced below. For simplicity, early sections of this guide refer to file names directly, not by DEFINE name or by reference to the startup sequence.

Using CLASS MAP DEFINEs

A DEFINE is a collection of attributes to which a common name has been assigned. These attributes can be passed to a process simply by referring to the DEFINE name from within the process. The =_DEFAULTS DEFINE is an example of such a DEFINE; this DEFINE passes the default node name, volume, and subvolume to a process.

The DEFINE mechanism can be used for passing file names to processes; this kind of DEFINE is called a CLASS MAP DEFINE. The following example creates a CLASS MAP DEFINE called =MYFILE and gives it a FILE attribute equal to \SWITCH.$DATA.MESSAGES.ARCHIVE:

```
1> SET DEFINE CLASS MAP, FILE \SWITCH.$DATA.MESSAGES.ARCHIVE
2> ADD DEFINE =MYFILE
```

Whenever your process accesses the DEFINE =MYFILE, it gets the name of the file specified in the DEFINE. For example, when your process opens =MYFILE, the file that actually gets opened is \SWITCH.$DATA.MESSAGES.ARCHIVE.

See Section 7, Using DEFINEs, for a complete discussion on how to use DEFINEs in your application programs.

Using the Startup Sequence

The startup sequence is a sequence of messages that are passed from the parent process to the new process when the process is created. The exchange of messages has to be agreed upon by both processes but typically involves passing a form of the IN and OUT file names in the Startup message, and sometimes other file names in Assign messages.

See Section 8, Communicating With a TACL Process, for information on how to access this information for processes that are started by the TACL process. For processes that are started from an application, see Section 16, Creating and Managing Processes.

Creating and Accessing Files

The rest of this section describes how to use Guardian procedures to perform common operations on files, such as creating, opening and closing, and reading and writing, as well as gathering information about files and handling file-system errors.

Creating Files

The technique for creating files depends on the type of file you are creating. You can create files interactively through the TACL command interpreter or certain utilities or programmatically by calling Guardian procedures. This guide is concerned with manipulating files programmatically. For details of the relevant command-interpreter commands, read the *Guardian User’s Guide*.

Disk files, for example, can be created programmatically using the FILE_CREATE[LIST]_ procedure or interactively using the TACL CREATE or File Utility Program (FUP) CREATE command. Device files are created either at operating-system generation time or by COUP (D-series releases) and SCF (G-series releases); they are not created programmatically. Process files are created when a process is created either programmatically using the PROCESS_LAUNCH_ or PROCESS_CREATE_ procedure or interactively using the TACL RUN command. One of the most important effects of creating a file is that a file name is given to the file.

Creating Disk Files

You can use either the FILE_CREATE_ or FILE_CREATELIST_ procedure to create disk files programmatically. FILE_CREATE_ allows you to specify most of the commonly used properties that a disk file can have, such as the file type (unstructured, relative, entry sequenced, or key sequenced), block length, record length, and extent sizes. Some files, however, need properties that you cannot assign using FILE_CREATE_ (such as alternate-key files and partitioned files); for these files, you need to use the FILE_CREATELIST_ procedure.

Some examples of what you can do with the FILE_CREATE_ procedure are given here. For specific examples of using FILE_CREATELIST_, see Section 5, Communicating With Disk Files. For complete details of both of these procedures, see the *Guardian Procedure Calls Reference Manual*.

The following lines of code create a permanent, unstructured disk file:

```
STRING NAME[0:ZSYS^VAL^LEN^FILENAME - 1];
STRING .S^PTR;
  .
  .
NAME ':=' "$OURVOL.MYSUBVOL.DATAFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @NAME;
ERROR := FILE_CREATE_(NAME:ZSYS^VAL^LEN^FILENAME,
                      LENGTH);
```

The first parameter to the call passes the name of the file to be created. In this case, the name is $OURVOL.MYSUBVOL.DATAFILE. Because the node name is not specified, the node name in the =_DEFAULTS DEFINE is used.

Note. File names should normally be passed to a process either in a DEFINE (see Section 7, Using DEFINEs) or in the Startup message (see Section 8, Communicating With a TACL Process). For simplicity, however, examples throughout this section receive hard-coded file names.

The first parameter also indicates the maximum length of the file name in bytes. The buffer (NAME in this example) should also have a length equal to the maximum file-name length. In this case, the literal ZSYS^VAL^LEN^FILENAME provided in the ZSYSTAL file has been used to reserve a buffer large enough for any file name including space for future expansion of file names. Here, the maximum length need only reserve enough space for the supplied file-name string, because the actual length of the file name is known on input.

The second parameter designates the actual length of the supplied file name. File names are variable length, so it is necessary to tell the operating system how many bytes to expect. In this case, pointers have been used to identify each end of the file-name string before computing the string length.

To create a temporary file, use the FILE_CREATE_ procedure without specifying the subvolume or file ID of the name. For example:

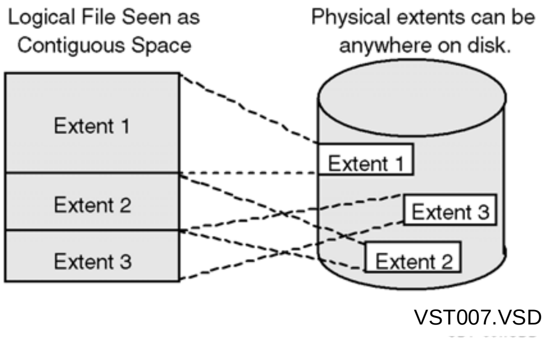
```
NAME ':=' "$OURVOL" -> @S^PTR;
LENGTH := @S^PTR '-' @NAME;
ERROR := FILE_CREATE_(NAME:ZSYS^VAL^LEN^FILENAME,
                      LENGTH);
```

Here, a temporary file is created somewhere on the volume $OURVOL. The name of the temporary file is returned in the NAME variable, and the name length in LENGTH. In this case, you should use the ZSYS^VAL^LEN^FILENAME literal to allow future expansion of the file name, because the length of the file name is not known on input.

Allocating Extents

So far no attention has been paid to how the operating system allocates disk space for a created file. It does so in **extents**, where an extent is a physically contiguous area of disk that may be as small as 2048 bytes or as large as 128 megabytes (MB). While applications see a file as a logically contiguous area of storage, the operating system splits the file space into extents. Figure 2-2 shows an example of a file split into three extents.

Figure 2-2. File Space Allocated in Extents


Logical File Seen as Contiguous Space
Physical extents can be anywhere on disk.
Extent 1
Extent 2
Extent 3
Extent 1
Extent 3
Extent 2
VST007.VSD

If you do not specify an extent size, the operating system uses the default extent size of one page (2048 bytes). Smaller extents mean less wasted allocated disk space, but CPU overhead is reduced by having larger extents because there are fewer extents to manage. A smaller extent size is therefore suitable for small files because it wastes less disk space. Larger files can be managed more efficiently with larger extents, because that results in fewer extents to manage.

Each file that is not empty is made up of at least one extent, the primary extent; an empty file has no extents. If a file is larger than the primary extent size, additional secondary extents are allocated. The secondary extents are all the same size, but the primary extent may be a different size than the secondary extents. Extents are automatically allocated to the file by the disk process as the need arises up to a file-dependent maximum value.

Parameters of the FILE_CREATE[LIST]_ procedure allow you to specify the extent sizes. One parameter specifies the length of the primary extent in pages (2048-byte units). Another parameter specifies the length of each secondary extent, also in pages.

The following example allocates a primary extent of 8 megabytes and secondary extents of 1 megabytes each.

```
PRIMARY^EXTENT^SIZE := 4096;
SECONDARY^EXTENT^SIZE := 512;
NAME := "$OURVOL.MYSUBVOL.DATAFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @NAME;
ERROR := FILE_CREATE_(NAME:ZSYS^VAL^LEN^FILENAME,
                      LENGTH,
                      !file^code!,             !not specified
                      PRIMARY^EXTENT^SIZE,
                      SECONDARY^EXTENT^SIZE);
```

So far you have seen how to control the size of the extents allocated to a file. You also need to set the amount of space that can be allocated to the file by specifying the maximum number of extents. By default, up to 16 extents can be allocated as needed.

You set the maximum number of extents initially using another parameter of the FILE_CREATE_ procedure. The following example sets the maximum to 32:

```
MAX^EXTENTS := 32;
CALL FILE_CREATE_(NAME:ZSYS^VAL^LEN^FILENAME,
                  LENGTH,
                  !file^code!,
                  PRIMARY^EXTENT^SIZE,
                  SECONDARY^EXTENT^SIZE,
                  MAX^EXTENTS);
```

This number can be changed either by using the FUP ALTER command—see the *File Utility Program (FUP) Reference Manual* —or programmatically using the SETMODE procedure call, function 92—see the *Guardian Procedure Calls Reference Manual*.

Creating Processes

You can create processes either by issuing the RUN command from the TACL command interpreter or by calling the PROCESS_LAUNCH_ or PROCESS_CREATE_ Guardian procedure from a program. (How to use the PROCESS_LAUNCH_ procedure is explained in Section 16, Creating and Managing Processes.)

The TACL RUN command can create named or unnamed processes. If you use the NAME option of the RUN command, then a named process is created. Without the NAME option, the RUN command usually creates an unnamed process, unless the RUNNAMED flag is specified for the object file, in which case the process is always named. (See the discussion earlier in this section about named and unnamed processes.)

If the process is created programmatically using the PROCESS_CREATE_ procedure, the process is named or unnamed depending on the information supplied with the call. One parameter of the PROCESS_CREATE_ procedure is known as the *`name-option`* parameter. If it is 1, then the process is named using the name supplied in the *`name`*`:`*`length`* parameter.

When creating processes from $SYSTEM.SYSnn, specifying the $SYSTEM.SYSTEM subvolume is recommended. When you specify the $SYSTEM.SYSTEM subvolume, the system dynamically searches for the object file first in the $SYSTEM.SYSTEM subvolume. If the object file is not found in $SYSTEM.SYSTEM, the search continues in the $SYSTEM.SYSnn subvolume. When you specify the $SYSTEM.SYSnn subvolume, you are explicitly specifying the location of the object. This means that any stored reference to the object must be updated to point to the location of the new object whenever a new version of the operating system is installed.

The following example starts a process from the program contained in the disk file \SYSTEM1.$MASTER.PROGS.SERVER and names the process $SER1. This example uses the literal ZSYS^VAL^PCREATOPT^NAMEINCALL supplied in the ZSYSTAL file to specify that the process will be named. This example also uses the ZSYS^VAL^LEN^PROCESSDESCR literal from the ZSYSTAL file to specify the maximum length of the returned process descriptor.

```
OBJECT^FILENAME ':=' "\SYSTEM1.$MASTER.PROGS.SERVER"
                      -> @S^PTR;
OBJFILENAME^LEN := @S^PTR '-' @OBJECT^FILENAME;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEINCALL;
PROCESS^NAME ':=' "$SER1" -> @S^PTR;
PROCESSNAME^LEN := @S^PTR '-' @PROCESS^NAME;
ERROR := PROCESS_CREATE_(
                     OBJECT^FILENAME:OBJFILENAME^LEN,
                     !library^file:lib^name^len!,
                     !swap^file:swap^name^len!,
                     !ext^swap^file:ext^swap^len!,
                     !priority!,
                     !processor!,
                     !process^handle!,
                     !error^detail!,
                     NAME^OPTION,
                     PROCESS^NAME:PROCESSNAME^LEN,
                     DESCR:ZSYS^VAL^LEN^PROCESSDESCR,
                     DESCLEN);
```

Your program can now send messages to $SER1 by opening and writing data to the process file name returned in the DESCR array variable.

As when creating any file, you need to supply the maximum file-name length. Again we recommend using the ZSYS^VAL^LEN^PROCESSDESCR literal from the ZSYSTAL file for this purpose. The actual length of the process descriptor is returned in the DESCLEN integer variable.

If the *`name-option`* parameter is set to 2, then the operating system provides a name. To set the *`name-option`* parameter to 2, we recommend using the ZSYS^VAL^PCREATOPT^NAMEDBYSYS literal from the ZSYSTAL file. In this case, the *`name`*`:`*`length`* parameter is omitted. A named-form process descriptor (a process file name without any qualifier) is returned in DESCR:

```
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEDBYSYS;
ERROR := PROCESS_CREATE_(
                     OBJECT^FILENAME:OBJFILENAME^LEN,
                     !library^file:lib^name^len!,
                     !swap^file:swap^name^len!,
                     !ext^swap^file:ext^swap^len!,
                     !priority!,
                     !processor!,
                     !process^handle!,
                     !error^detail!,
                     NAME^OPTION,
                     !name:length!,
                     DESCR:ZSYS^VAL^LEN^PROCESSDESCR,
                     DESCLEN);
```

If *`name-option`* is set to 0, then an unnamed process descriptor is returned in DESCR. You can make sure that the *`name-option`* parameter is correctly set by using the ZSYS^VAL^PCREATOPT^NONAME literal:

```
NAME^OPTION := ZSYS^VAL^PCREATOPT^NONAME;
ERROR := PROCESS_CREATE_(
                     OBJECT^FILENAME:OBJFILENAME^LEN,
                     !library^file:lib^name^len!,
                     !swap^file:swap^name^len!,
                     !ext^swap^file:ext^swap^len!,
                     !priority!,
                     !processor!,
                     !process^handle!,
                     !error^detail!,
                     NAME^OPTION,
                     !name:length!,
                     DESCR:ZSYS^VAL^LEN^PROCESSDESCR,
                     DESCLEN);
```

Opening Files

Your program must open a file before gaining access to it. Use the FILE_OPEN_ procedure to open any file on your system or network. You supply the procedure with a file name and the name of a variable in which to return the file number. You will later use this file number to perform operations on the open file. The association of the file number with the file name remains until the file is closed.

The FILE_OPEN_ procedure call has many options; only the most common are described here. For a complete description of all FILE_OPEN_ parameters, refer to the *Guardian Procedure Calls Reference Manual*.

Examples of opening disk files, device files, and process files follow.

Opening Disk Files

To open a disk file, use a call like the following:

```
FILE^NAME ':=' "$OURVOL.MYSUBVOL.DATAFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM);
```

The first parameter (FILE^NAME:LENGTH) is the file name created by the FILE_CREATE[LIST]_ procedure (or the TACL CREATE command or the FUP CREATE command). LENGTH is an integer variable that specifies the length in bytes of the file name. If the file was created using the FILE_CREATE_ procedure, you can use the length value returned by that procedure.

The second parameter (FILENUM) returns a number that your program uses to identify the file in subsequent operations. Once the file is opened, you use this number to identify the file.

Opening Disk Files for Ensured Data Integrity

To ensure data integrity when you perform write operations to a disk file, you need to open that file using a nonzero value for the *sync-depth* parameter as follows:

```
FILE^NAME ':=' "$OURVOL.MYSUBVOL.DATAFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
SYNC^DEPTH := 1;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    SYNC^DEPTH);
```

Setting the *sync-depth* parameter to a nonzero value causes the disk I/O process to checkpoint information to its backup process when performing I/O operations. If the CPU on which the primary disk process is running fails, then the backup disk process can use the checkpointed information to establish whether it needs to complete the operation or whether the operation finished successfully before the failure occurred. Recovery from a CPU failure in this way is invisible to the application process.

If you do not set the *sync-depth* parameter to a nonzero value on opening the file, the backup disk process has no way of knowing whether the operation finished successfully. If you open a disk file with a zero sync depth, then a CPU failure could cause corruption of data.

Opening Devices

Opening a device file is similar to opening a disk file. The call to FILE_OPEN_ is the same; the only difference is in determining the file name. Remember that device naming is a system-management function, therefore you need to know some system-configuration information before attempting to open a device file.

The following example opens a printer called $LP1:

```
FILE^NAME ':=' "$LP1" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM);
```

The next example opens the home terminal of the process:

```
ERROR := PROCESS_GETINFO_(
                         !process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMINAL^NAME:ZSYS^VAL^LEN^FILENAME,
                         LENGTH);
ERROR := FILE_OPEN_(TERMINAL^NAME:LENGTH,
                    FILENUM);
```

Here, the PROCESS_GETINFO_ call returns the name of the home terminal in the variable TERMINAL^NAME, along with the file-name length in LENGTH. Both of these values are supplied to the FILE_OPEN_ call. You now use the returned file number to perform I/O operations on the terminal.

Opening Processes

To open a process, you simply pass the process file name and its length to the FILE_OPEN_ procedure. If the process you are opening was created by the current process (using, for example, a call to PROCESS_CREATE_ as described earlier in this section), then you use the process descriptor returned by the process creation procedure. If the process was created outside the current process, then you can pass the process name in the FILE_OPEN_ call.

Consider a requester process $REQ that needs the services of a server process $SER1 that was created and named using the RUN command. The requester may open the server process as follows:

```
FILE^NAME ':=' "$SER1" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM);
```

To receive messages sent to it, the $SER1 process must open its $RECEIVE file:

```
FILE^NAME ':=' "$RECEIVE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
RECEIVE^DEPTH := 1;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    RECEIVE^DEPTH);
```

The requester process can now pass messages to the server process.

Reading and Writing Data

The operating system supports several procedure calls that enable reading and writing of files:

- READX and READ each read a record from an open file into an application buffer. There are two differences between these procedures: one is that READX can read into a buffer in either the user data segment or an extended data segment, whereas READ must have the application buffer in the user data segment; the other difference is that READX reads data into a buffer declared with data type STRING, whereas READ reads data into a buffer declared with data type INT.
- WRITEX and WRITE each write a record to an open file from an application buffer. The same two differences apply to WRITE and WRITEX as to READ and READX, this time regarding the position and data type of the buffer containing the data to be written.

See Section 17, Managing Memory, for discussions on accessing extended data segments.

Before performing any I/O to a file, the file must be open as described in Opening Files, earlier in this section.

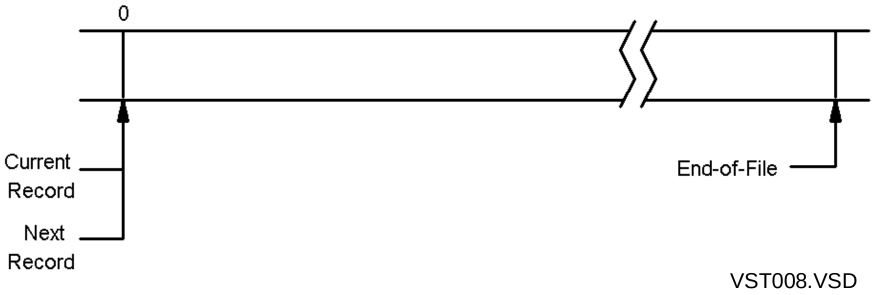
I/O With Disks

I/O to all supported types of disk files is described in detail in Section 5, Communicating With Disk Files. This subsection describes some of the simple operations that can be performed on unstructured files. The various pointers used to control position within the file and the procedures used to perform I/O operations are introduced.

Associated with each file that you have open are three pointers: a current-record pointer, a next-record pointer, and an end-of-file pointer. The current-record pointer and next-record pointer are used to establish at which byte in the file a read or write operation will begin. These pointers can be manipulated to perform sequential or random access.

The end-of-file pointer simply points to the last relative byte of the file plus 1. It is automatically advanced by the number of bytes written whenever you append data to the file.

When you open a file, the current-record and next-record pointers point to the first byte in the file:



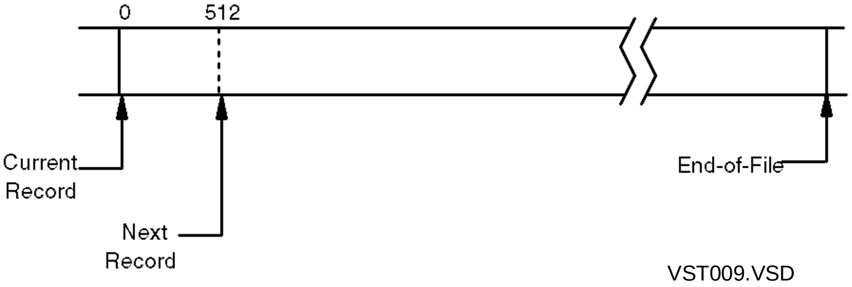
A READ[X] or WRITE[X] procedure call always begins at the byte pointed to by the next-record pointer. The next-record pointer is advanced on each READ[X] or WRITE[X] call to provide automatic sequential access to the file.

Normally, the next-record pointer is rounded up to an even value. However, if the file was created as an odd-unstructured file (by setting bit <15> of the FILE_CREATE_ *`options`* parameter to 1) then the next-record pointer is advanced by exactly the number of bytes transferred.

Following the read or write operation, the current-record pointer indicates the first byte affected by the read or write operation. The following example transfers 512 bytes of data from the disk file starting at relative byte 0 into a buffer in memory called SBUFFER.

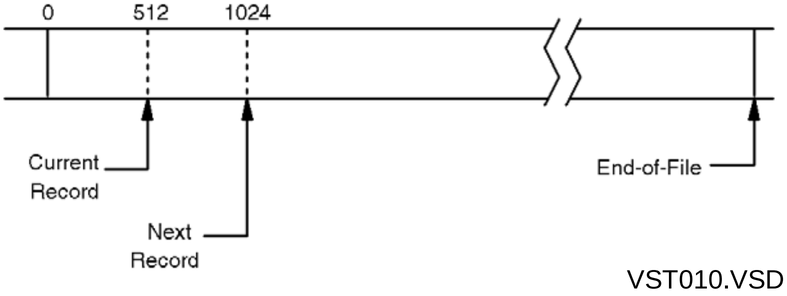
```
STRING .SBUFFER[0:511];
  .
  .
RCOUNT := 512;
CALL READX(FILENUM,
           SBUFFER,
           RCOUNT,
           NUMXFERRED);
```

The actual number of bytes transferred is returned in NUMXFERRED. The positions of the pointers are as follows. The next-record pointer is increased by 512 bytes; the current-record pointer still addresses relative byte 0:


0
512
Current Record
Next Record
End-of-File
VST009.VSD

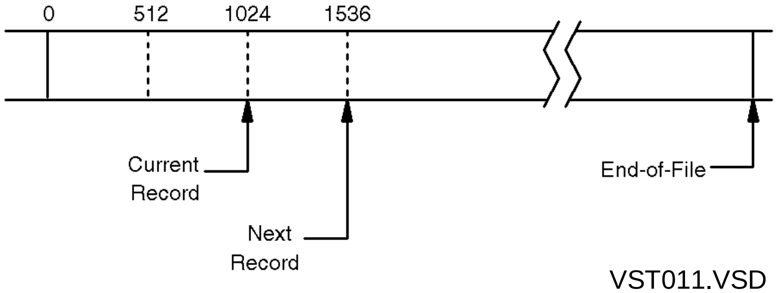
If you reissue an identical READX call, the next 512 bytes are read into SBUFFER (starting at byte 512). The next-record pointer is increased by 512 bytes and now points to relative byte address 1024; the current-record pointer points to relative byte 512:

```
RCOUNT := 512;
CALL READX(FILENUM,
           SBUFFER,
           RCOUNT,
           NUMXFERRED);
```


0
512
1024
Current Record
Next Record
End-of-File
VST010.VSD

If you now issue the following WRITEX call, 512 bytes are written into the disk file, starting at the byte addressed by the next-record pointer. The effect on the pointers is the same as if you had issued a READ call:

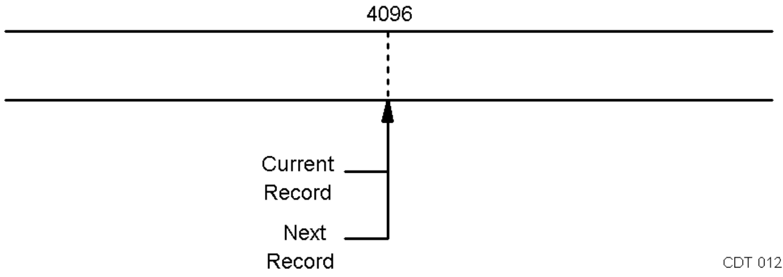
```
WCOUNT := 512;
CALL WRITEX(FILENUM,
            SBUFFER,
            WCOUNT);
```



VST011.VSD

Random access to a disk file is provided by the POSITION procedure. This procedure sets the current-record pointer and next-record pointer. The following example sets both these pointers to relative byte 4096:

```
FILE^POINTERS := 4096D;
CALL POSITION(FILENUM,
              FILE^POINTERS);
```



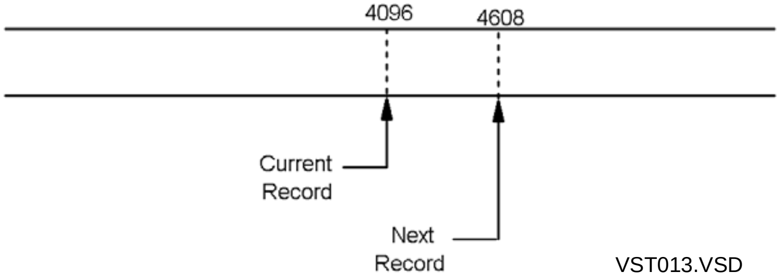
CDT 012CDD

Note that the procedure requires a double-length integer.

A READX call now reads the data, starting at relative byte 4096:

```
RCOUNT := 512;
CALL READX(FILENUM,
           SBUFFER,
           RCOUNT,
           NUMXFERRED);
```

This call transfers 512 bytes from the disk file starting at relative byte 4096 into SBUFFER. The next-record pointer is increased by 512 bytes so that further sequential access is automatic. The current-record pointer still points at relative byte 4096:

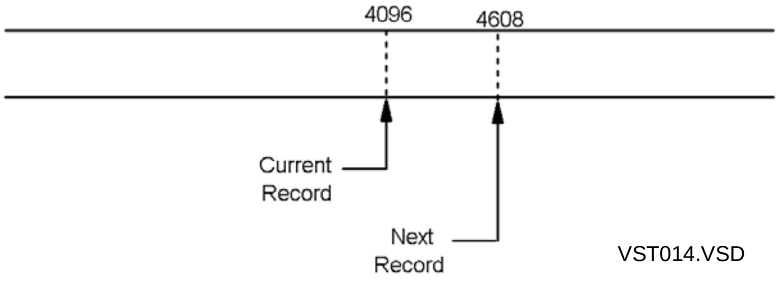


I/O operations can also be performed starting at the relative byte pointed at by the current-record pointer. To read from the current-record pointer, you use the READUPDATE[X] procedure; to write starting at the current-record pointer, you use the WRITEUPDATE[X] procedure.

A typical read record, update record, write record back sequence makes use of a READ[X] call followed by a WRITEUPDATE[X] call. For example, if you follow the above READX call with a WRITEUPDATEX call that uses the same buffer size as the READX call, then the record read by the READX call gets written over because the WRITEUPDATEX call starts writing at the current-record pointer, not the next-record pointer:

```
WCOUNT := 512;
CALL WRITEUPDATEX(FILENUM,
                  SBUFFER,
                  WCOUNT);
```

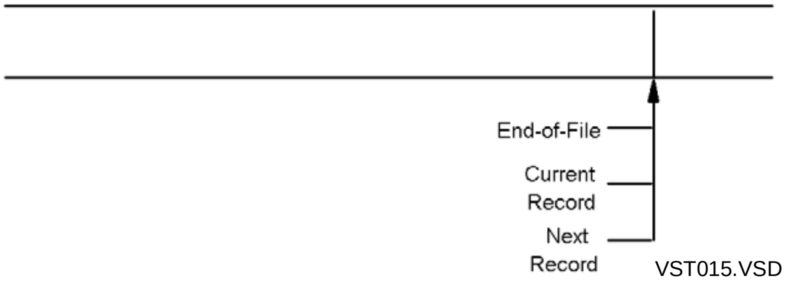
Following the WRITEUPDATEX call, the current-record and next-record pointers remain unchanged:



To append records to a file, you must position the current-record and next-record pointers to the end of the file. You do this by supplying the POSITION procedure with -1 as the byte address:

```
FILE^POINTERS := -1D;
CALL POSITION(FILENUM,
              FILE^POINTERS);
```

Following the above call, the current-record and next-record pointers are positioned as follows:



Successive write operations then append records to the file.

I/O With Devices

Sections 9 through 12 describe I/O operations to device files in detail. This subsection briefly presents the procedures used and gives one specific example of the WRITEREADX procedure that is particularly useful for communicating with terminals.

Because devices can be treated as files, input and output to devices can be done using read and write operations like those for disk files. For device-specific operations, such as setting the mode of operation for a device, you use the SETMODE and CONTROL procedures.

Writing to a printer involves simply using the WRITE[X] procedure. Communicating with magnetic tape uses the READ[X] and WRITE[X] procedures along with the CONTROL procedure that is used to space the tape backwards and forwards. I/O to terminals can also be done using simple READ[X] and WRITE[X] calls. In many applications, however, communicating with a terminal involves displaying a prompt and then waiting for a response. The WRITEREAD[X] procedure combines both of these operations into one procedure.

The WRITEREAD[X] procedure has two parts: the first part writes the contents of a memory buffer to the specified file, and the second reads the response back into the same buffer. The procedure requires at least four parameters: the file number of the file you want to communicate with, the buffer name, the number of bytes to be written, and the maximum number of bytes that will be returned. When communicating with a terminal in conversational mode, the read ends when a line-termination character is entered (typically a carriage return). A fifth parameter returns the actual number of bytes read.

The following example prompts the user to enter an account number. The procedure returns when the user has entered a number and pressed the line-termination character:

```
SBUFFER ':=' "PLEASE ENTER ACCOUNT NUMBER:  " -> @S^PTR;
WCOUNT := @S^PTR '-' @SBUFFER;
RCOUNT := 72;
CALL WRITEREADX(FILENUM,
                SBUFFER,
                WCOUNT,
                RCOUNT,
                NUMXFERRED);
```

The call writes 30 bytes from the memory buffer SBUFFER, then prepares for reading up to 72 bytes of information back into the same buffer. A count of the number of bytes entered is given in NUMXFERRED.

I/O With Processes

A process writes messages to another process by writing to the open process file. To read messages sent by another process, your process must read from its $RECEIVE file. (By default, messages from the operating system are also read through $RECEIVE; you can choose not to receive file management system messages, however, by setting the appropriate bit in the FILE_OPEN_ procedure *`options`* parameter.)

Communication between processes can be two-way or one-way. In two-way communication, the first process sends a message to the second process, and then the second process reads the message and responds with reply information. In one-way communication, one process simply sends a message to the other and the other process reads it; the second process passes no information in the response to the first process.

Consider a requester process $REQ that performs two-way communication with a server process $SER1. $REQ opens $SER1 and $SER1 opens $RECEIVE. Because $REQ wants to read a reply from $SER1, it sends a request message using the WRITEREADX procedure. Because the server expects to send reply text or an error indication back to the requester, it reads the message from $RECEIVE using a READUPDATEX call and then sends a reply using a REPLYX call.

```
         $REQ                        $SER1

NAME ':=' "$SER1";               NAME ':=' "$RECEIVE";
LEN := 5;                        LEN := 8;
ERROR := FILE_OPEN_(             ERROR := FILE_OPEN_(
         NAME:LEN,                 NAME:LEN,
         FN,                       FN,
         !access!,                 !access!,
         !exclusion!,              !exclusion!,
         !nowait^depth!,           !nowait^depth!,
         1);                       1);
 .                                .
 .                                .
BUFF ':=' "MESSAGE...";           .
CALL WRITEREADX(FN,BUFF,         CALL READUPDATEX(FN,BUFF,
         WCOUNT,RCOUNT);                       RCOUNT);
                                 .
                                 .
                                 .
                                 CALL REPLYX(BUFFER,COUNT);
```

The call to REPLYX by the server satisfies the WRITEREADX call. That is, whatever REPLYX returns in its BUFFER is what WRITEREADX reads.

Note that the sixth parameter, the receive depth, is specified in the FILE_OPEN_ call in the server. Here, the receive depth is specified as 1 to enable the READUPDATEX procedure to process one message at a time. The receive depth is discussed in detail in Section 6, Communicating With Processes, along with other interprocess communication issues.

In one-way communication, the server passes no information in the response to the requester. In this case, the requester can issue the request using the WRITEX procedure instead of WRITEREADX. Because the server does not send any information in the reply, it can read the message from $RECEIVE using the READX procedure. The reply to the requester is made when the READX finishes, allowing the WRITEX in the requester to finish. If there is no message in $RECEIVE, the READX call waits until a message arrives (unless the “nowait” option is specified; see Section 4, Using Nowait Input/Output).

```
         $REQ                            $SER1

NAME ':=' "$SER1";               NAME ':=' "$RECEIVE";
LEN := 5;                        LEN := 8;
ERROR := FILE_OPEN_(             ERROR := FILE_OPEN_(
         NAME:LEN,                 NAME:LEN,
         FNUM,                     FNUM,
         !access!,                 !access!,
         !exclusion!,              !exclusion!,
         !nowait^depth!,           !nowait^depth!,
         1);                       1);
 .                                .
 .                                .
 .                                .
BUFFER ':=' "MESSAGE...";
CALL WRITEX(FNUM,BUFFER,         CALL READX(FNUM,BUFFER,
     WCOUNT);                         RCOUNT);
 .                                .
 .                                .
```

There is actually a third way of communicating with another process (sometimes called “one-and-a-half-way communication”) that has elements of one-way communication and two-way communication. Here, the requester sends a message to the server using the WRITE[X] procedure (not expecting return data). If the server reads the message using the READUPDATE[X] procedure, the WRITE[X] does not terminate until the server responds by calling REPLY[X]. The WRITE[X] procedure cannot read data, but it does return the file-system error number sent in the reply.

Getting File Information

The following related procedures provide information on all files: disk files, device files, and process files:

FILE_GETINFO_	Returns brief information about an open file identified by file number.
FILE_GETINFOBYNAME_	Returns brief information about a file identified by file name. The file need not be open to get information using this procedure.
FILE_GETINFOLIST_	Returns extended information about an open file identified by file number.
FILE_GETINFOLISTBYNAME_	Returns extended information about a file identified by file name.

Refer to the *Guardian Procedure Calls Reference Manual* for a complete description of each of these procedures. This guide presents a brief overview.

Information provided by the brief-form procedures includes:

- The name of the file and file-name length
- The last error number returned from the file system
- Device type and subtype, as well as information about the specific device type
- The physical record length associated with the file

The extended-form procedures can return all the above, plus information about the current position pointers, key values, access modes, exclusion modes, and so on.

One common use of the FILE_GETINFO_ procedure is to return the value of the last file-system error. File-system errors are discussed in the next subsection.

Handling File-System Errors

An error number is associated with the completion of each procedure call to the file system. The error number indicates whether the procedure executed successfully. If the procedure did not execute successfully, then you can use the error number to help determine what went wrong.

An error number is a 16-bit signed integer. To avoid using negative numbers, only 15 bits are used, yielding a range of error numbers from 0 up to about 32K. Error numbers are categorized as follows:

- Error number 0 indicates that the procedure executed successfully.
- Error numbers in the range 1 through 9 are warnings. Warnings indicate that some event has happened that may or may not be harmful to your process. For example, reaching the end of file returns a warning error number.

- Error numbers in the range 10 and up indicate an error encountered in a standard operation, such as an attempt to access a file before it is open, or that a system component failed while the procedure was executing.
- Error numbers 300 through 511 are reserved for application-dependent use.

Returned Error Numbers and Condition Codes

Some file-system procedures return the error number directly to the calling program. Others return only a condition code as follows:

>	condition-code-greater-than (CCG) indicates a warning
<	condition-code-less-than (CCL) indicates an error
=	condition-code-equal (CCE) indicates successful execution

Following procedure calls that provide only a condition code, your program must issue a FILE_GETINFO_ call to obtain the error number.

Your program should always check for errors immediately after executing a file-system procedure call. If the call returns the error itself, simply check the return value. If a nonzero error number is returned, your program could, for example, call a user-written procedure to process the error. The following example calls the procedure FILE^ERRORS to process the error number:

```
FILE^NAME ':=' "$OURVOL.MYSUBVOL.DATAFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    FILENUM);
IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);
```

If the procedure sets a condition code, you need to call the FILE_GETINFO_ procedure to determine the error number. The error number is returned in the second parameter:

```
CALL WRITEX(FILENUM,BUFFER,WCOUNT);
IF <> THEN
BEGIN
   CALL FILE_GETINFO_(FILENUM,ERROR);
   CALL FILE^ERRORS(ERROR);
END;
```

The *Guardian Procedure Calls Reference Manual* indicates which procedures return the error number and which procedures set a condition code.

The procedure FILE^ERRORS might, for example, simply print the error number on the terminal. The user would then be expected to look up the error number using the ERROR command. The sample program at the end of this section shows a procedure coded to work this way. Alternatively, FILE^ERRORS could be coded to communicate directly with the ERROR program, causing the error text to be displayed on the terminal screen without the need for manual intervention. Section 20, Interfacing With the ERROR Program, describes how to do this.

Retrying After an Error

In some cases, the error condition may be temporary. Your program can try the operation again after a period of time or following some operator intervention. For example, the following errors typically indicate a temporary error in an operation that your program can retry once the condition that caused the error is corrected:

Error 40	Operation timed out
Error 73	File/record locked
Error 100	Device not ready or controller not operational
Error 101	No write ring (magnetic tape)
Error 102	Paper out, bail open, or end of ribbon (line printer)
Error 110	Only BREAK access permitted (terminal)
Error 111	Terminal operation aborted because of BREAK

It might be useful to retry errors 101 and 102 more than once.

In other cases, the error number indicates a condition that typically cannot be recovered by trying the operation again, as in the following examples:

Error 11	File not in directory or record not in file, or the specified tape file is not present on a labeled tape
Error 12	File in use
Error 14	Device does not exist
Error 43	Unable to obtain disk space for file extent
Error 45	File is full
Error 48	Security violation
Error 49	Access violation, or attempt to use an unexpired labeled tape for output, or mismatch between DEFINE USE attribute (input or output/extend) and the current operation (read or write)

If you do choose to retry the operation that caused one of these errors, be sure to delay for an appropriate period between detecting the error and retrying the operation. You should also keep a retry counter or a timer to indicate when to give up retrying the operation. See Section 18, Managing Time, for information about setting up timers.

For details of all file-system errors, including a discussion of the cause of the error, the action taken by the system, and suggested action for your program to take, see the *Guardian Procedure Errors and Messages Manual*.

Closing Files

You can close files explicitly using the FILE_CLOSE_ procedure.

```
ERROR := FILE_CLOSE_(FILENUM);
```

If you do not explicitly close a file, the file remains open until the process stops. When a process stops, all files that the process has open are automatically closed.

Once you have closed a file, the file number can no longer access that file. The file number is now available to be reassigned to another file.

Accessing Files: An Example

The following simple program uses many of the procedure calls described in this section. The program shows communication with a terminal and with an unstructured disk file.

The program is designed to keep a daily log of comments. It allows the user to append comments to the log or read comments from the log.

The program prompts a user to request one of these functions:

- Append a record to the disk file. Records are 512 bytes long and are terminated when the line-termination character is entered.
- Read a record from the disk file and display it on the terminal. Read operations begin at the first record in the file. The program prompts the user to make additional read requests. Successive read operations display records sequentially.
- Exit the program.

Before running the program, the data file to contain the log must exist. You can create this file either programmatically by using the FILE_CREATE_ procedure as described earlier in this section or interactively using either the CREATE command or the FUP CREATE command. The following example uses the FUP CREATE command:

```
1> FUP
-CREATE $ADMIN.OPERATOR.LOGFILE
CREATED - $ADMIN.OPERATOR.LOGFILE
-EXIT
2>
```

The program consists of the following procedures:

- The LOGGER procedure is the main procedure. It calls INIT to handle the Startup messages and open files. It calls the GET^COMMAND procedure to prompt the user for the function to perform and then calls the appropriate procedure to execute the selected function. If the user selected “r,” the LOGGER procedure calls READ^RECORD. If the user selected “a,” the LOGGER procedure calls APPEND^RECORD. If the user selected “x,” the LOGGER procedure calls EXIT^PROGRAM.

- The INIT procedure reads and discards the Startup messages before opening the terminal file and the disk file containing the daily log.
- The GET^COMMAND procedure displays a menu of options on the user's terminal and returns the selected option ("r," "a," or "x") to the main procedure.
- The READ^RECORD procedure reads records from the log file. Starting from the beginning of the file, this procedure reads each record from the file, displays it on the terminal, and then prompts the user to read the next record. If the user declines or the end of the file is reached, the procedure returns to the main procedure.
- The APPEND^RECORD procedure prompts the user to enter some comments and then writes those comments to the end of the file.
- The EXIT^PROGRAM procedure stops the program.
- The ILLEGAL^COMMAND procedure responds to the user entering an illegal function. That is, the user entered something other than "r," "R," "a," "A," "x," or "X." After informing the user of the illegal input, the procedure returns to the main procedure.
- The FILE^ERRORS^NAME and FILE^ERRORS procedures display error messages when the program receives a file-system error on trying to execute a file-system procedure call. FILE^ERRORS^NAME is used if the file is not yet open. FILE^ERRORS is used if the file is already open. After displaying a file-system error message, these procedures stop the process.

Note. Near the beginning of the source code that follows are some definitions of TAL DEFINEs used by the program to help formatting and displaying messages. Refer to the *TAL Reference Manual* for details of TAL DEFINEs. Do not confuse TAL DEFINEs with the file system DEFINEs described in Section 7, Using DEFINEs.

The TAL code for this program appears on the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;   !max-file name
                                           ! length
LITERAL BUFSIZE = 512;

STRING .SBUFFER[0:BUFSIZE];       !I/O buffer (one extra char)
STRING .S^PTR;                    !pointer to end of string
INT     LOGNUM;                   !log file number
INT     TERMNUM;                  !terminal file number

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_)
?LIST

!-------------------------------------------------------------
!  These DEFINEs make it easier to format and print messages.
!-------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE  PUT^INT(N) =

         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

    DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

    DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

    DEFINE  PRINT^STR(S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name and its length
! and the error number.  This procedure is used when the
! file is not open, so there is no file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN-1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;


!-------------------------------------------------------------
! Procedure to prompt the user for the next function to be
! performed:
!
! "r" to read records
! "a" to append a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'a' for Append to Log, ");
   PRINT^STR("     'x' for Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r." The start of the read is selected randomly by record
! number.  The user has the option of sequentially reading
! subsequent messages.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT    COUNT^READ;
   INT    ERROR;

!  Position current-record and next-record pointers
!  to the beginning of the file:

   CALL POSITION (LOGNUM, 0D);
   IF <> THEN  CALL FILE^ERRORS (LOGNUM);

!  Loop reading and displaying records until user
!  declines to read next record (any response other than
!  "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Read a record from the log file and display
   !  it on the terminal.  Display "No such record"
   !  if reach end of file:

      CALL READX(LOGNUM,SBUFFER,BUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(LOGNUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(LOGNUM);
      END;

      CALL WRITE^LINE(SBUFFER,COUNT^READ);

      PRINT^BLANK;
      !  Prompt the user to read the next record.  The user
      !  must respond "y" to accept, otherwise the procedure
      !  returns to select next function:

         SBUFFER ':=' ["Do you want to read another ",
                                    "record (y/n)? "]
                       -> @S^PTR;
         CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                         BUFSIZE,COUNT^READ);
         IF <> THEN CALL FILE^ERRORS(TERMNUM);

         SBUFFER[COUNT^READ] := 0;
      END
      UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
   END;


   !------------------------------------------------------------
   ! Procedure for appending a record.  The user selected
   ! function "a." The user is prompted to enter comments.  The
   ! procedure puts the comments in a new record at the end of
   ! the file.
   !------------------------------------------------------------

   PROC APPEND^RECORD;
   BEGIN
      INT    COUNT^READ;

      PRINT^BLANK;

   !  Prompt user for comments and read comments into the
   !  buffer:

      SBUFFER ':=' "Enter today's comments:  "
                    -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   !  Blank out portion of buffer past last character read:

      SBUFFER[COUNT^READ] ':=' " " & SBUFFER[COUNT^READ]
                               FOR BUFSIZE-COUNT^READ BYTES;

   !  Place the next-record pointer at the end of file and
   !  write the new record there:

      CALL POSITION (LOGNUM,-1D);
      IF <> THEN  CALL FILE^ERRORS(LOGNUM);

      CALL WRITEX(LOGNUM,SBUFFER,BUFSIZE);
      IF <> THEN CALL FILE^ERRORS(LOGNUM);
   END;
```

```
!------------------------------------------------------------
! Procedure to exit the program.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "a," or "x."
!------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and
!  return to prompt again for a valid function:

   PRINT^STR ("INVALID COMMAND: "  &
              "Type either 'r,' 'a,' or 'x'");
END;
```

```
!------------------------------------------------------------
! Procedure to initialize the program. It calls
! INITIALIZER to dispose of the startup sequence of messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .LOGNAME[0:MAXFLEN - 1];      !name of log file
   INT     LOGLEN;                      !length of log name
   STRING .TERMNAME[0:MAXFLEN - 1];     !terminal file
   INT     TERMLEN;                     !length of term name
   INT     ERROR;

!  Read and discard the startup sequence of messages.

   CALL INITIALIZER;

!  Open the terminal file.  For simplicity this program uses
!  the home terminal; the recommended approach is to use the
!  IN file read from the Startup message; see Section 8 for
!  details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,
                       TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the log file with a sync depth of 1:

   LOGNAME ':=' "$XCEED.DJCEGD10.LOGFILE" -> @S^PTR;
   LOGLEN := @S^PTR '-' @LOGNAME;
   ERROR := FILE_OPEN_(LOGNAME:LOGLEN,
                       LOGNUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN, ERROR);
END;
```

```
!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then  goes into a loop calling GET^COMMAND
! to get the next user request and then calling a procedure
! to carry out the selected request.
!-------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function "x":

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command:

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "a", "A"  -> CALL APPEND^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
   END;
END;
```

3

Coordinating Concurrent File Access

Several processes can access the same file at the same time. This section describes the procedures that allow you to coordinate such concurrent access. Each process indicates (when opening the file) how it intends to use the file, either by specifying the access mode and the exclusion mode to the file or by accepting default values.

Topics covered in this section include:

- How to set the access mode for a file; the access mode limits the operations to be performed by the opener. The access mode is specified as read/write, read-only, or write-only. Setting the Access Mode, below, provides details.
- How to set the exclusion mode; the exclusion mode specifies how much access other processes are allowed. It can provide shared, protected, or exclusive access. Setting the Exclusion Mode, later in this section, shows how to do this.
- How to apply a lock to a file that is already open. In addition to exclusion specified at file-open time, the file system also allows you to apply a lock to a file that is already open. You do this using the LOCKFILE procedure as described in Locking a File in this section. You can also inadvertently cause your process to wait indefinitely because it has to wait for a locked resource that never becomes available. The last subsection, Avoiding Deadlocks, describes how to prevent this.

Locking issues regarding concurrent access at the record level, are not described here; see Section 5, Communicating With Disk Files, for details. This section discusses file-level concurrency issues.

Setting the Access Mode

When you open a file, you do so with an access mode that indicates what kind of operations you will perform on the file once it is open. The access mode can allow you to read and write to the file, perform only read operations, or perform only write operations.

The third parameter of the FILE_OPEN_ procedure (the *access* parameter) specifies the access mode. This parameter can have one of the following values:

0	Read/write access (the default access)
1	Read-only access
2	Write-only access
3	Extend access (applies to magnetic tape only—refer to Section 12, Communicating With Magnetic Tape)

The following example opens three files, one for reading and writing, one for read-only access, and one for write-only access:

```
LITERAL READ^WRITE = 0;
LITERAL READ^ONLY = 1;
LITERAL WRITE^ONLY = 2;
  .
  .
  .
ERROR := FILE_OPEN_(FILENAME1:LENGTH1,
                    FILENUM1,
                    READ^WRITE);
IF ERROR <> 0 THEN ...

ERROR := FILE_OPEN_(FILENAME2:LENGTH2,
                    FILENUM2,
                    READ^ONLY);
IF ERROR <> 0 THEN ...

ERROR := FILE_OPEN_(FILENAME3:LENGTH3,
                    FILENUM3,
                    WRITE^ONLY);
IF ERROR <> 0 THEN ...
```

Whether access to the file is granted, however, depends on file ownership and on the security assigned to the file by the owner. The file owner has the right to determine who can open the file and for what purpose. The file owner determines who is allowed to read from the file, write to the file, execute the file, and purge the file. Access to the file for any of these purposes can be limited to the file owner, the group or network community the owner belongs to, or all users of the system or network. (Access can be controlled at a finer-grained level if the file is Safeguard protected.)

If access to the file is refused because of security, the FILE_OPEN_ procedure returns an error.

The output of the TACL FILEINFO command shows the security assigned to each file. The columns headed “RWEP” show the security assigned for reading, writing, executing, and purging, respectively, as follows:

O	Only the file owner on the local node
U	Only the file owner on the local node or a remote node
G	Users in the same group as the owner and on the local node
C	Users in the same group as the owner on the local node or a remote node
–	Access by local super-ID user only
A	All users on the local node
N	All users on the local node or any remote node

To access any file on a remote node requires matching remote passwords on the local node and the remote node.

Refer to the *TACL Reference Manual* for more details on the TACL FILEINFO command.

Consider a file owned by user 24,48 (where 24 is the user's group number and 48 is the user number within that group) with security permissions "GOA-":

G	In the read column indicates that anyone in the user's group on the local node can read the file, but no one outside the group can read it.
O	In the write column says only the owner on the local node can open the file for writing.
A	In the execute column says anyone on the local node can execute the file.
–	In the purge column says only the super-ID user on the local node can purge the file.

A process executed by user 24,40 now tries to open the file. If this process tries to open the file for reading and writing, the open will fail because the file security permits only the owner to write to the file. Similarly, the open will fail if user 24,40 tries to open the file in write-only mode. If user 24,40 tries to open for reading only, however, the open will succeed because the owner and the opener are in the same group and the owner has set the security on the file to allow anyone in the same group to read the file.

Note. If files are protected by Safeguard, then the FILEINFO command does not return useful information. If the Safeguard protection is applied at the file level, then FILEINFO returns a string of asterisks for the security permissions for that file. If the Safeguard protection is applied at the subvolume or volume level, then the output of the FILEINFO command appears as a normal FILEINFO display but does not reflect the Safeguard protection.

The Safeguard protection mechanism is different from the mechanism described here. See the *Safeguard Reference Manual* for details.

For details on how to set and change the security of a file using the TACL program, refer to the *Guardian User's Guide*.

Setting the Exclusion Mode

To ensure a consistent view of data, it is often necessary to restrict concurrent access to a file. The file system provides the following levels of exclusion:

- Shared (the default exclusion)

  The opening process tolerates any access mode of other openers, but no other process is allowed exclusive access to the file. Shared mode also prevents another process from opening the file with the protected mode if this process has the file open for writing.

  Shared mode permits the highest level of concurrent operation and is therefore the normal mode of operation in a multiple-user environment where there is access to a shared database. Any transaction-processing application would be typical. Here, data integrity can be provided at a lower level (for example, the record level).

- Protected

  The opening process tolerates only other openers with read-only access mode. In addition, processes attempting to open the file for exclusive access are barred from the file. Different processes may have the same file open in protected mode, but only if all openers are opening the file for read-only access.

  Protected mode is used when a consistent view of the entire database is required, such as for end-of-period stock taking or balance sheet preparation.

- Exclusive

  The opening process allows no other access to the file until the file is closed.

  Use exclusive mode only when no other access can be tolerated; for example, during major restructuring of your database.

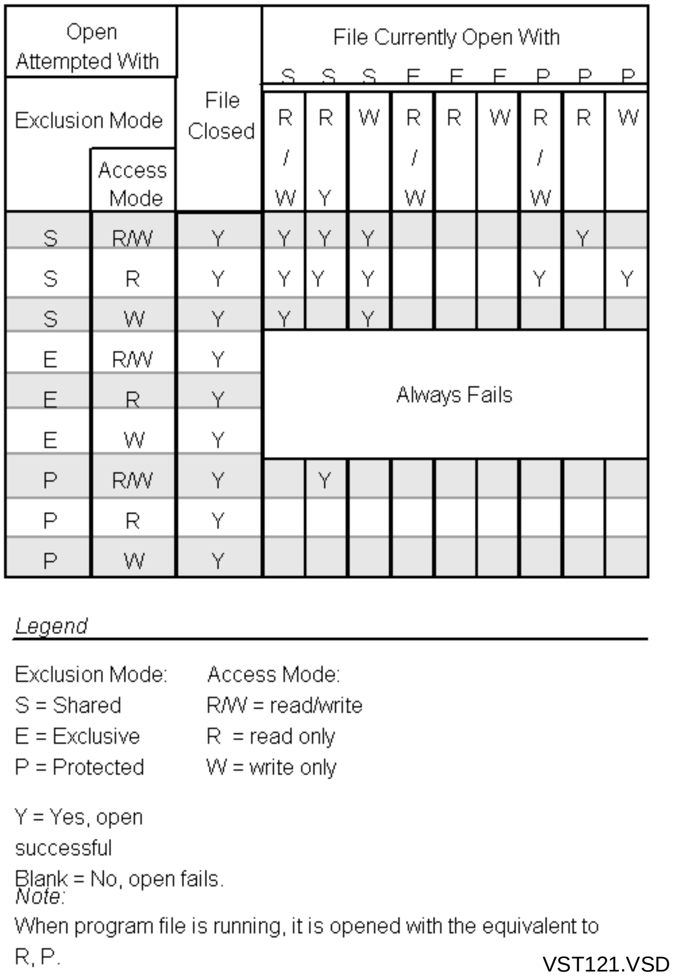
Figure 3-1 summarizes the effects of all possible combinations of access mode and exclusion mode. When you read the table, the exclusion and access mode with which some other process has the file open are given along the top of the table. The exclusion and access mode you are requesting to open the file with are shown in the leftmost columns. “Y” at the intersection indicates that the new open is allowed and the requested permissions granted.

You use the *`exclusion`* parameter of the FILE_OPEN_ procedure to specify the exclusion mode. This parameter can have the following values:

0	Shared access
1	Exclusive access
3	Protected access

If the parameter is omitted, 0 (shared mode) is assumed by default.

Figure 3-1. Exclusion and Access Mode Compatibility



Open Attempted With		File Closed	File Currently Open With								
Exclusion Mode	Access Mode		S	S	S	E	E	E	P	P	P
			R/W	R Y	W	R/W	R	W	R/W	R	W
S	R/W	Y	Y	Y	Y					Y	
S	R	Y	Y	Y	Y				Y		Y
S	W	Y	Y		Y						
E	R/W	Y	Always Fails								
E	R	Y									
E	W	Y									
P	R/W	Y		Y							
P	R	Y									
P	W	Y									

Legend

Exclusion Mode:	Access Mode:
S = Shared	R/W = read/write
E = Exclusive	R = read only
P = Protected	W = write only

Y = Yes, open successful

Blank = No, open fails.

Note:

When program file is running, it is opened with the equivalent to R, P.

VST121.VSD

The following example opens three files:

```
LITERAL READ^WRITE = 0;
LITERAL READ^ONLY = 1;
LITERAL WRITE^ONLY = 2;
LITERAL SHARED^ACCESS = 0;
LITERAL EXCLUSIVE^ACCESS = 1;
LITERAL PROTECTED^ACCESS = 3;
  .
  .
  .
ERROR := FILE_OPEN_(FILENAME1:LENGTH1,FILENUM1);
IF ERROR <> 0 THEN ...
```

```
ERROR := FILE_OPEN_(FILENAME2:LENGTH2,FILENUM2,
                    READ^ONLY,
                    PROTECTED^ACCESS);
IF ERROR <> 0 THEN ...
```

```
ERROR := FILE_OPEN_(FILENAME3:LENGTH3,
                    FILENUM3,
                    !access!,
                    EXCLUSIVE^ACCESS);
IF ERROR <> 0 THEN ...
```

The first FILE_OPEN_ call uses the default values to open FILENAME1 for reading and writing with shared exclusion mode. The second call opens FILENAME2 for read-only access with protected exclusion mode. The last call opens FILENAME3 for reading and writing (by default) but with exclusive access.

If the open cannot proceed due to an exclusion mode held by another process, then the FILE_OPEN_ procedure returns an error.

Locking a File

So far this guide has discussed methods of exclusion that are applied when a file is opened. You can apply a temporary exclusion to a file by locking it with the LOCKFILE procedure. While you have the file locked, no other process can access any part of the locked file (with one exception described later in this subsection). The lock can be removed using the UNLOCKFILE procedure:

```
CALL LOCKFILE(FILENUM);
  .
  .
  .
CALL UNLOCKFILE(FILENUM);
```

If your process tries to lock a file that is locked by another process, your call to LOCKFILE does not finish until the other process unlocks the file. If your process tries to write to a locked file, the write operation fails and an error code is returned. If your process tries to read from a locked file, the read operation does not finish until the other process unlocks the file.

You can use function 4 of the SETMODE procedure to change the processing that occurs when you try to lock or read from a file that is locked by another process. SETMODE function 4 allows several such options. For example:

- You can specify that lock and read operations on files that are locked by another process should finish immediately and return with an error indication:

  ```
  LITERAL SET^LOCK^MODE = 4,
          REJECT^MODE   = 1;
    .
    .
  CALL SETMODE(FILENUM,
               SET^LOCK^MODE,
               REJECT^MODE);
  ```

- You can specify that read operations should finish normally and that data should be returned in the buffer even though another process has the file locked. You should be aware, of course, that the process that has locked the file might change the record after the process reads the record:

```
LITERAL READ^THROUGH^MODE = 2;
  .
  .
CALL SETMODE(FILENUM,
             SET^LOCK^MODE,
             READ^THROUGH^MODE);
```

Refer to the *Guardian Procedure Calls Reference Manual* for a complete description of all SETMODE function 4 options.

You can obtain information about file locks, such as how many processes hold locks on the file and how many processes are waiting for file locks, by calling the FILE_GETLOCKINFO_ procedure. Refer to the *Guardian Procedure Calls Reference Manual* for details of this procedure.

Note.

2 You can apply locks to records as well as files. You apply a lock to a record using the LOCKREC procedure and remove the lock using the UNLOCKREC procedure. You can manipulate record locks in all the ways described above for file locks. See Section 5, Communicating With Disk Files, for more information on disk-file records.

2 Throughout this section, the default of waited I/O has been assumed. In cases where the process is described as waiting for some kind of response, the rules might change if nowait I/O is used. Nowait I/O is described in the next section.

Avoiding Deadlocks

There are two kinds of deadlocks your process might encounter:

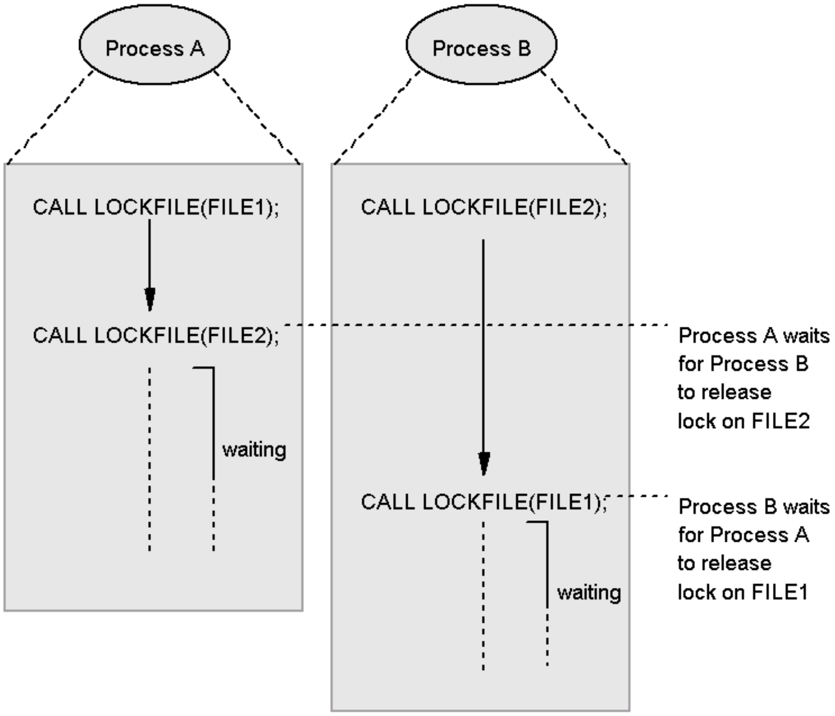
- Multiple-process deadlocks
- Single-process deadlocks

The following paragraphs describe each kind of deadlock and how to detect or avoid them.

Avoiding Multiple-Process Deadlocks

Figure 3-2 shows an example of how two processes competing for resources are able to cause each other to wait indefinitely. This kind of situation is known as a deadlock.

Figure 3-2. Two Processes in Deadlock

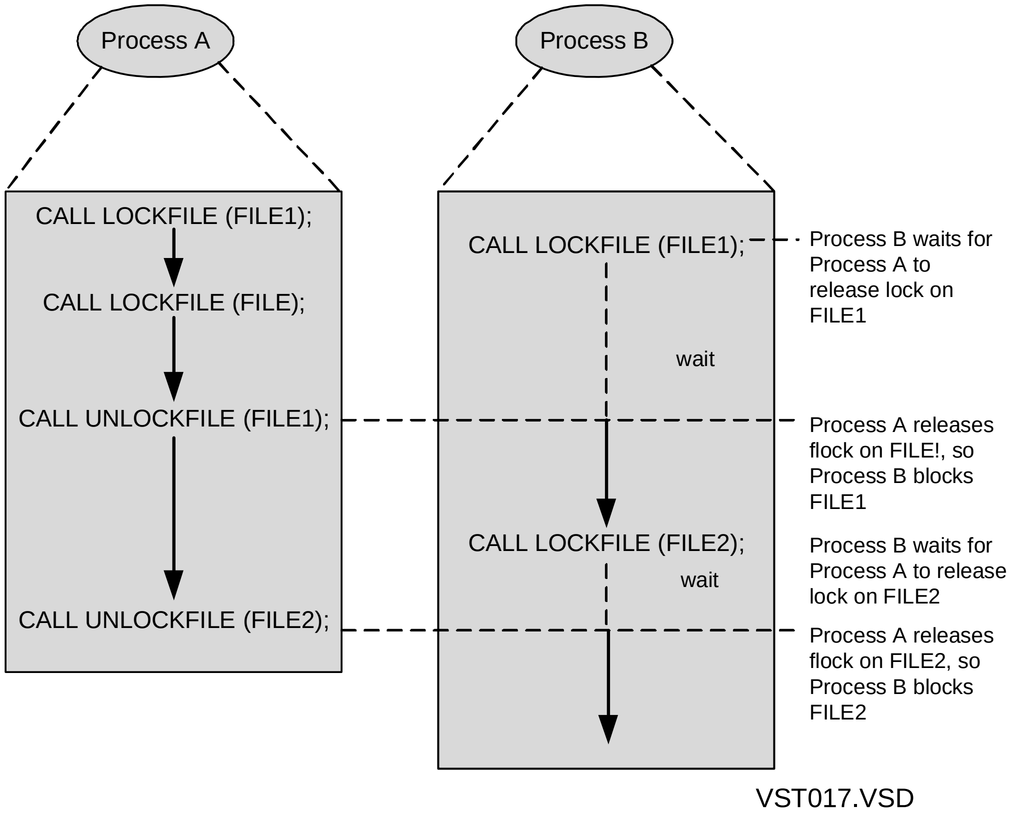

Process A
Process B
CALL LOCKFILE(FILE1);
CALL LOCKFILE(FILE2);
waiting
CALL LOCKFILE(FILE2);
CALL LOCKFILE(FILE1);
waiting
Process A waits for Process B to release lock on FILE2
Process B waits for Process A to release lock on FILE1

VST016.VSD

Process A acquires the lock on file 1. Process B acquires the lock on file 2. Now process A would like to lock file 2 but cannot because process B has it locked. Process B can never release the lock it has on file 2 because it is waiting for a lock on file 1, which process A can never release.

You can avoid this kind of deadlock by careful programming practice, as shown in Figure 3-3.

Figure 3-3. Avoiding the Two-Process Deadlock


Process A
Process B
CALL LOCKFILE (FILE1);
CALL LOCKFILE (FILE);
CALL UNLOCKFILE (FILE1);
CALL UNLOCKFILE (FILE2);
CALL LOCKFILE (FILE1);
wait
CALL LOCKFILE (FILE2);
wait
Process B waits for Process A to release lock on FILE1
Process A releases flock on FILE!, so Process B blocks FILE1
Process B waits for Process A to release lock on FILE2
Process A releases flock on FILE2, so Process B blocks FILE2
VST017.VSD

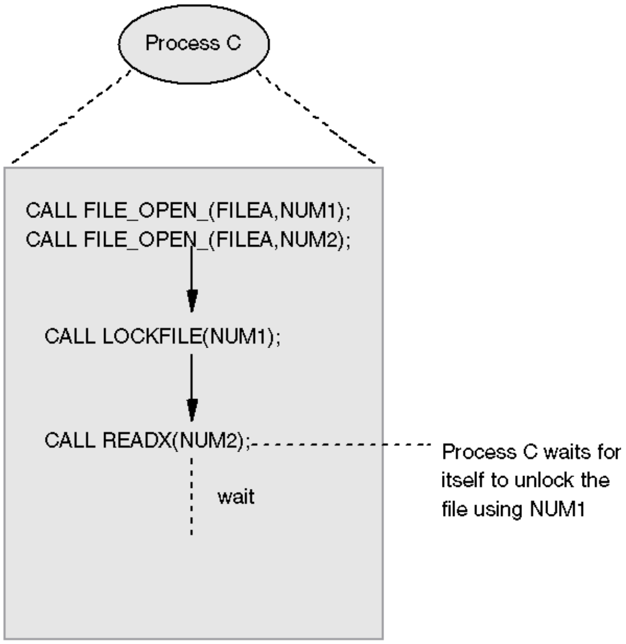
By making sure that each process acquires its locks in the same order, you ensure that no deadlock can occur. Here, process B waits when it tries to lock file 1. Process A is then able to get both the locks it needs to continue. Process A eventually releases its locks, allowing process B to continue.

Note that more than two processes and two files may be involved in this kind of deadlock situation. Process A may wait for process B, which waits for process C, which waits for process D, which waits for process A. The solution is the same. Always acquire the locks in the same order in each process.

Avoiding Single-Process Deadlocks

A process can also cause itself to deadlock, as shown in Figure 3-4.

Figure 3-4. Single-Process Deadlock


Process C
CALL FILE_OPEN_(FILEA,NUM1);
CALL FILE_OPEN_(FILEA,NUM2);
CALL LOCKFILE(NUM1);
CALL READX(NUM2);
wait
Process C waits for itself to unlock the file using NUM1

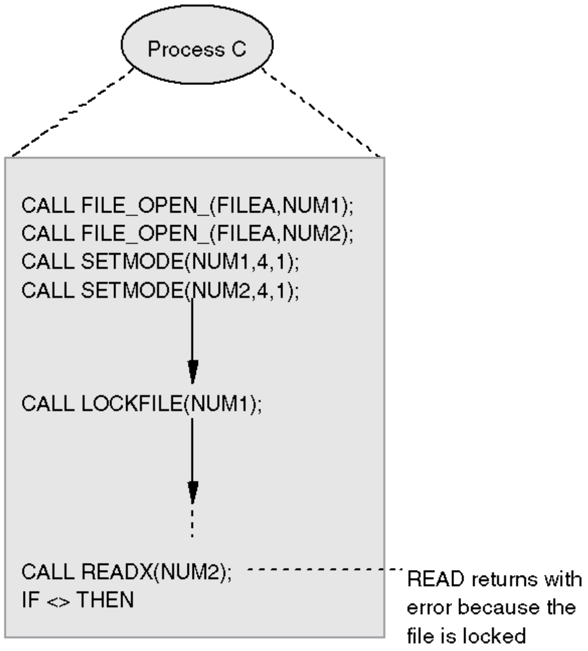
VST018.VSD

Here, process C has opened the same file twice, returning two file numbers. The process acquires a lock using one of the file numbers, then tries to read the file using the other file number. Process C waits forever for itself to unlock the file.

Note. This kind of deadlock does not occur if the file is protected by the NonStop Transaction Manager/MP (TM/MP), because TM/MP organizes locks by transaction ID, not file number. See the *NonStop TM/MP Application Programmer’s Guide* for details.

Correct use of SETMODE function 4 allows your program to avoid this kind of deadlock. Figure 3-5 shows how.

Figure 3-5. Avoiding the Single-Process Deadlock



```
CALL FILE_OPEN_(FILEA,NUM1);
CALL FILE_OPEN_(FILEA,NUM2);
CALL SETMODE(NUM1,4,1);
CALL SETMODE(NUM2,4,1);

CALL LOCKFILE(NUM1);

CALL READX(NUM2);
IF <> THEN
```

READ returns with error because the file is locked

VST019.VSD

By issuing a call to SETMODE function 4 on each file number, subsequent read operations return immediately with an error code if the read could not proceed. Deadlock is thus avoided.

4 Using Nowait Input/Output

This section discusses how to do I/O operations without having the process wait for completion of the operation. A process that does not wait for I/O operations to finish is said to be using **nowait I/O**.

This section uses examples to show the different ways in which you can use nowait I/O. It discusses how to write programs that:

- Perform a single nowait I/O operation against just one file
- Perform multiple I/O operations that run concurrently against just one file
- Perform multiple I/O operations that run concurrently against more than one file

A complete program is included that shows how to use nowait I/O to time out I/O operations.

Overview of Nowait Input/Output

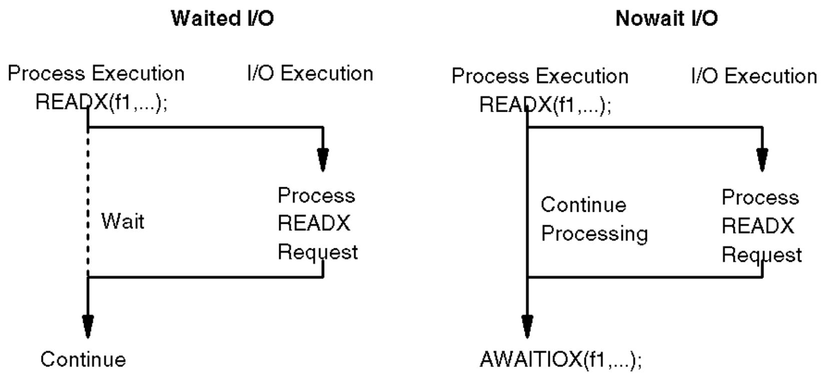
Normally, when a process issues an I/O request, the process waits for the operation to finish before continuing. The process enters the wait state (see Section 16, Creating and Managing Processes, for a discussion of process states) and some other process gains access to the CPU. Instead of having your process wait for the operation to finish, however, you can write your program to initiate the I/O operation and then continue processing while the I/O operation finishes. Completion of the I/O is detected later by a call to the AWAITIO[X] procedure.

You should use the AWAITIOX procedure to complete nowait READX, WRITEX, and WRITEREADX calls. You should use the AWAITIO procedure to complete nowait READ, WRITE, and WRITEREAD calls.

You can also use the FILE_COMPLETE_ procedure to complete nowait calls that you would otherwise complete by calling AWAITIO[X]. The FILE_COMPLETE_ procedure has special features; for example, you can use it to complete nowait I/O on any file from a predefined set of files, which can include both Guardian files and Open System Services (OSS) files. The FILE_COMPLETE_ procedure and its companion procedures, FILE_COMPLETE_SET_ and FILE_COMPLETE_GETINFO_, are discussed later in this section.

Figure 4-1 compares waited I/O with nowait I/O.

Figure 4-1. Waited and Nowait I/O


Waited I/O
Process Execution
I/O Execution
READX(f1,...);
Wait
Process
READX
Request
Continue
Nowait I/O
Process Execution
I/O Execution
READX(f1,...);
Continue
Processing
Process
READX
Request
AWAITIOX(f1,...);

VST020.VSD

When you use nowait I/O, however, you typically do so for one of the following reasons:

- To apply a time limit to an operation
- To support multiple independent logical processing threads in a single program

The ability to overlap application processing with I/O operations is often secondary.

To use nowait I/O, you need to do the following:

- Set the *`nowait`* parameter in the FILE_OPEN_ call to specify nowait I/O on all operations to the file that use the returned file number.
- Use calls to the AWAITIO[X] procedure to check for or wait for completion of the I/O operation.

Note. It is important to distinguish between the FILE_OPEN_ `nowait` parameter and the nowait bit in the FILE_OPEN_ `option` parameter. The `nowait` parameter allows you to establish nowait access to the file once the file is open. The nowait bit in the `option` parameter allows you to perform the open operation itself in a nowait manner. All combinations are possible in the same call except that you cannot have a nowait open operation and permit waited I/O operations on the same file.

I/O operations initiated by the procedures listed below execute in parallel with your process when invoked on a file opened for nowait I/O. You must complete each of these calls by a separate call to the AWAITIO[X] procedure if the I/O operation is against a file opened for nowait I/O.

CONTROL	SETMODENOWAIT (for other than disk files)
CONTROLBUF	
LOCKFILE	UNLOCKFILE
LOCKREC	UNLOCKREC
READ[X]	WRITE[X]
READLOCK[X]	WRITEREAD[X]
READUPDATE[X]	WRITEUPDATE[X]
READUPDATELOCK[X]	WRITEUPDATELOCK[X]

Note. When performing a nowait write operation, it is important that you do not overwrite your write buffer between the WRITE[X] or WRITEREAD[X] procedure call and the corresponding AWAITIOX procedure call. If you do modify the buffer in this period, then you cannot be sure which data actually gets written.

Applying a Nowait Operation on a Single File

The simplest case of a nowait operation is that of a single I/O operation against one file. In other words, multiple I/Os are not permitted to run concurrently against this file, nor are nowait operations performed against other files.

First of all, you need to open the file for nowait I/O. You do this by putting a nonzero value in the fifth parameter (*nowait*) of the FILE_OPEN_ procedure call. The *nowait* parameter specifies how many outstanding nowait I/O operations can concurrently exist against the file when identified by the returned file number. To allow only one nowait operation at a time, set this value to 1 as shown in the example below:

```
NOWAIT^DEPTH := 1;
ERROR := FILE_OPEN_(DATAFILE:LENGTH,
                    FILENUM,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...
```

After opening the file, you can issue an I/O operation against the file:

```
BYTES := 512;
CALL WRITEX(FILENUM,
            BUFFER,
            BYTES);
IF <> THEN ...
```

In this example, if the WRITEX call initiates just one I/O operation (as, for example, when writing to an unstructured file), it returns immediately to the program. If the WRITEX call initiates several I/O operations (as, for example, when writing to a key-sequenced file that has alternate keys), then the program waits until the last I/O operation starts. In either case, once control is returned to your program, the program continues to execute until it reaches a call to the AWAITIOX procedure. Additional attempts to initiate I/O operations on this file before the current I/O finishes will return an error because only one I/O is permitted to run at a time.

You must complete the I/O operation at some point later in the program. The AWAITIO[X] procedure gives you three ways you can do this, depending on the value you assign to the *`timeout`* parameter:

- Wait indefinitely until the I/O finishes. You do this by assigning -1D to the *`timeout`* parameter or by omitting the *`timeout`* parameter. In this case, AWAITIO[X] waits as long as it takes for the I/O to finish:

  ```
  CALL AWAITIOX(FILENUM);
  IF <> THEN ...
  ```

- Specify a time limit for the I/O to finish. Set the *`timeout`* parameter to the number of one hundredths of a second that your program will wait for the operation to finish. Error 40 (operation timed out) is returned if the operation does not finish within the time limit, and the operation is canceled.

  ```
  LIMIT := 100D;
  CALL AWAITIOX(F1,
                !buffer^address!,
                !count^transferred!,
                !tag!,
                LIMIT);
  IF <> THEN ...
  ```

△ **Caution.** When the file system cancels an operation due to timeout, it does not undo those parts of the operation that have already finished. For example, in an operation that requires multiple physical I/O operations, some of those operations may have finished and others may not have finished.

- Return immediately to the program whether the I/O operation finishes or not. You do this by setting the time limit to zero:

  ```
  LIMIT  := 0D;
  CALL AWAITIOX(F2,
                !buffer^address!,
                !count^transferred!,
                !tag!,
                LIMIT);
  ```

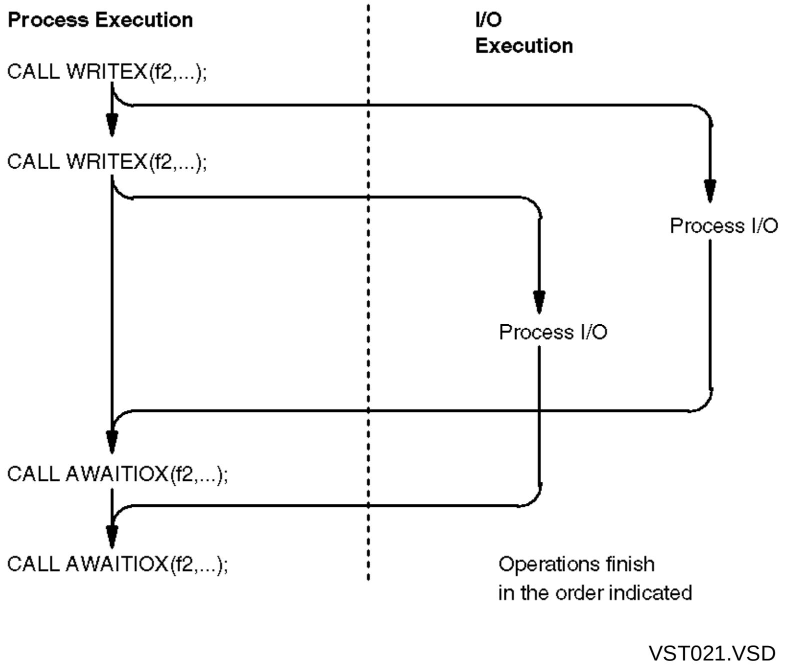
  This method effectively checks whether the I/O is finished and then returns. The procedure call returns error 40 (operation timed out) if the I/O is not finished, but it does **not** cancel the operation.

  This example should be part of a loop that regularly checks for I/O completion.

Applying Multiple Nowait Operations on a Single File

Files other than disk files allow multiple I/O operations to run concurrently. Figure 4-2 shows several I/O operations executing concurrently against the same file. This file must be opened specifically to allow concurrent I/O operations.

Figure 4-2. Multiple Concurrent Operations on One File


Process Execution
I/O Execution
CALL WRITEX(f2,...);
CALL WRITEX(f2,...);
Process I/O
Process I/O
CALL AWAITIOX(f2,...);
CALL AWAITIOX(f2,...);
Operations finish in the order indicated
VST021.VSD

To permit multiple operations on the same file to run concurrently, you must open the file with the *`nowait`* parameter set to the maximum number of concurrent I/Os that you will permit against that file.

Once several concurrent I/O operations have started, you can require that operations finish in the order in which they started, or you can allow them to finish in any order. SETMODE function 30 determines which of these alternatives you use. The way you distinguish between operations depends on which option you choose.

- To force I/O operations to finish in the order in which you start them, you make no call to SETMODE function 30 or you set the *`parameter-1`* parameter to 0 before calling SETMODE function 30. The first call to AWAITIO[X] completes the oldest outstanding operation, the second call completes the second oldest, and so on.
- To allow completions to occur in any order, you must call SETMODE function 30 with bit 15 of the *`parameter-1`* parameter set to 1.

This subsection discusses both of these approaches.

In both of these approaches, the AWAITIO[X] call can wait indefinitely, time out, or check for completion and return immediately, exactly as in the single-I/O single-file model discussed in the previous subsection. (When a timeout expires, only the oldest I/O operation is canceled.)

Completing I/Os in the Order Initiated

The following example uses nowait I/O to start several I/O operations. In this case, calls to the AWAITIOX procedure complete the operations in the order they started.

```
NOWAIT^DEPTH := 4;
ERROR := FILE_OPEN_(PROCESSNAME:LENGTH,
                    F4,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...
 .
 .
 .
BYTES := 10;
LENGTH := 512;
CALL WRITEREADX(F4,BUFFER1,BYTES,LENGTH);
IF <> THEN ...

BYTES := 25;
LENGTH := 512;
CALL WRITEREADX(F4,BUFFER2,BYTES,LENGTH);
IF <> THEN ...

BYTES := 12;
LENGTH := 512;
CALL WRITEREADX(F4,BUFFER3,BYTES,LENGTH);
IF <> THEN ...
 .
 .
 .
CALL AWAITIOX(F4);
IF <> THEN ...

CALL AWAITIOX(F4);
IF <> THEN ...

CALL AWAITIOX(F4);
IF <> THEN ...
```

The FILE_OPEN_ call sets the nowait depth to 4, allowing up to four nowait operations to run concurrently against the file. The program then issues three WRITEREADX operations against the file and continues processing. Finally, the program issues three calls to the AWAITIOX procedure. The first call completes the operation started by the first WRITEREADX call, the second AWAITIOX call completes the operation started by the second WRITEREADX call, and so on.

Completing I/Os in Any Order

Now consider what happens when I/O operations are allowed to finish in any order. Two different uses of the SETMODE procedure produce slightly different results:

- Setting `parameter-1` to 1 causes I/O operations to finish in any order, except if more than one operation is ready to finish at the time of the AWAITIO[X] call. In this case, the operations that are ready finish in their issued order.
- Setting `parameter-1` to 3 causes I/O operations to finish in the order chosen by the operating system to be most efficient.

The next example shows how to use function 30 of the SETMODE procedure to permit I/O operations to finish in any order. Previous examples have simply passed the file number to the AWAITIOX procedure to identify the completing operation. The file number is enough to identify the I/O operation if only one such operation per file is allowed to run concurrently with the program. When you permit more than one operation per file to run concurrently, however, you need to initiate each operation with a unique tag so that you can identify the operation when it finishes. The following example shows this method.

```
INT(32) TAG, TAG1 := 1D, TAG2 := 2D, TAG3 := 3D;
LITERAL    CHOOSE^ORDER = 30,
           ANY^ORDER    = 1;
  .
  .
NOWAIT^DEPTH := 4;
ERROR := FILE_OPEN_(PROCESSNAME:LENGTH,
                    F4,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...
CALL SETMODE(F4,CHOOSE^ORDER,ANY^ORDER);
  .
  .
  .
BYTES := 10;
LENGTH := 512;
CALL WRITEREADX(F4,
                BUFFER1,
                BYTES,
                LENGTH,
                TAG1);
IF <> THEN ...
```

```
BYTES := 25;
LENGTH := 512;
CALL WRITEREADX(F4,
                BUFFER2,
                BYTES,
                LENGTH,
                TAG2);
IF <> THEN ...

BYTES := 12;
LENGTH := 512;
CALL WRITEREADX(F4,
                BUFFER3,
                BYTES,
                LENGTH,
                TAG3);
IF <> THEN ...
 .
 .
 .
CALL AWAITIOX(F4,
              !buffer^address!,
              BYTE^COUNT,
              TAG);
IF <> THEN ...
```

The sixth parameter (*`tag`*) of each WRITEREADX call assigns a unique tag. The AWAITIOX procedure retrieves the tag value in its own *`tag`* parameter and thereby identifies the completing operation. All procedures that are affected by nowait I/O can set the value of the tag.

The above example shows only one AWAITIOX call. You typically place this call in a loop that repeats for each initiated I/O.

Using File-System Buffering

HP recommends that you use a different application buffer for each concurrent read operation. However, if you must use the same buffer, then you need to use an intermediate file-system buffer to prevent the read operations from corrupting each other’s buffered data. You make use of file-system buffers by issuing SETMODE function 72.

△ **Caution.** You should not use file-system buffering with WRITE or WRITEREAD operations, because you can still corrupt your data when the file system performs implicit retry operations.

For files opened with the FILE_OPEN_ procedure, the operating system normally transfers data directly between the application buffer and the I/O buffer. If concurrent operations use the same application buffer, it is possible that one such operation overwrites the buffer before some other operation has completed its transfer. The result is that one of these operations transfers corrupted data.

By issuing SETMODE function 72, you cause the operating system to use an intermediate file-system buffer in the process file segment (PFS):

```
LITERAL USE^PFS = 72,
        ON      =  1;
 .
 .
CALL SETMODE(FILE^NUM,
             USE^PFS,
             ON);
```

Consider two concurrent read operations that use the same buffer without using file-system buffering:

1. The first read operation starts and reads data into the application buffer.
2. The second read operation starts and also reads data into the application buffer.
3. The AWAITIO[X] procedure now completes the first read operation. However, instead of returning the record read by the first operation, the buffer contains the record read by the second operation.

The same two read operations making use of file-system buffering in the PFS execute as follows:

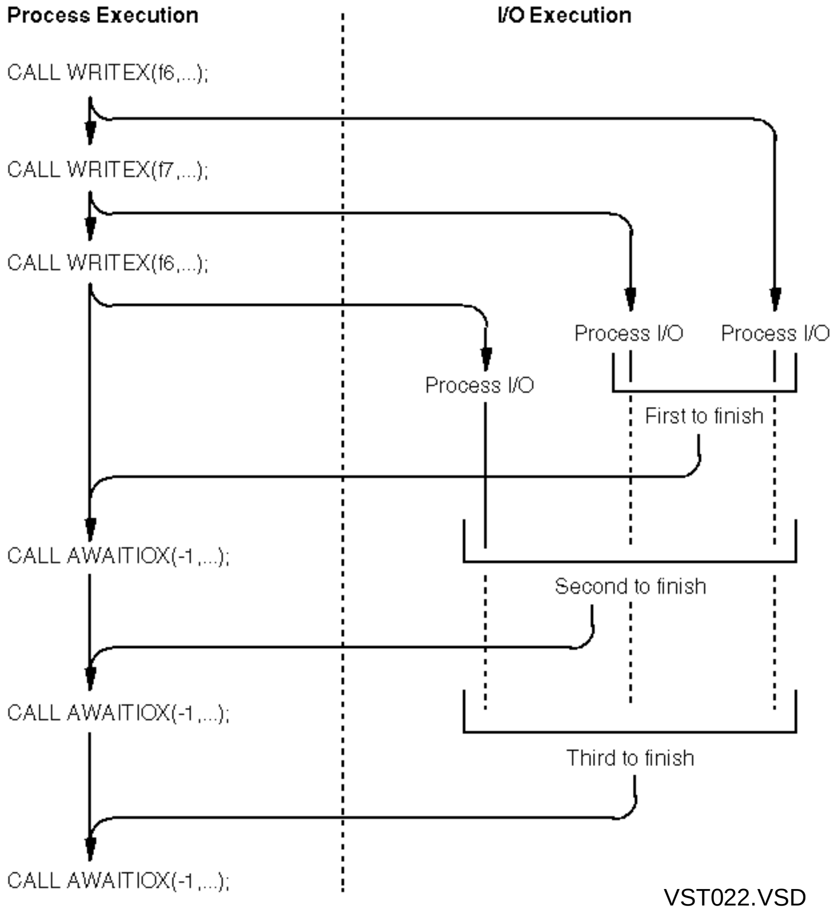
1. The first read operation starts and reads data into a location in the PFS.
2. The second read operation starts and reads data into a different location in the PFS.
3. The AWAITIO[X] procedure completes the first read, transferring data from the location in the PFS assigned to the first read operation. The returned data therefore corresponds to the first read operation.

Note. The most effective way to prevent concurrent I/O operations from destroying the contents of each other’s buffers is by using different buffer areas. That way there is no need for system buffering, and your program can take advantage of the efficiency of transferring data directly from the I/O buffers to the application buffers.

Applying Nowait Operations to Multiple Files

Your program may open several files for nowait I/O and issue one or more I/O operations against each file. (Some kinds of opens, for example those of disk files, allow only one operation at time.) Figure 4-3 shows how nowait operations can occur concurrently on multiple files.

Figure 4-3. Nowait Operations on Multiple Files


Process Execution
I/O Execution
CALL WRITEX(f6,...);
CALL WRITEX(f7,...);
CALL WRITEX(f6,...);
Process I/O
Process I/O
Process I/O
First to finish
CALL AWAITIOX(-1,...);
Second to finish
CALL AWAITIOX(-1,...);
Third to finish
CALL AWAITIOX(-1,...);
VST022.VSD

When nowait I/O operations are applied to multiple files, you cannot predict the order in which the operations will finish. In Figure 4-3, the first and third write operations are to the same files, therefore the first write will finish before the third write. The second write is to a different file; therefore it is not clear when it will finish with respect to the first and third writes.

To allow completion of a nowait operation to any file, you use the AWAITIO[X] procedure in a way that responds to the first completed I/O, regardless of the file that the I/O operation was made against. You do this by setting the `file-number` parameter of the AWAITIO[X] procedure to -1. When AWAITIO[X] returns, it sets the *`file-number`* parameter to the number of the file whose I/O operation finished.

If you issue just one I/O operation at a time to each file, then the file number returned by the AWAITIO[X] procedure is enough to identify the completed operation. However, it is often easier to use tags as described in the previous subsection.

A typical use for executing concurrent I/O operations against more than one file might be when a process communicates with more than one terminal. The problem with using waited I/O is that your program might, for example, issue a WRITEREADX call to the terminal of a user who has left the office. The user of the other terminal is locked out by this action, because the program waits indefinitely on the WRITEREADX call. Using nowait I/O, the WRITEREADX call can be issued to both terminals; the process responds to the terminal that replies:

```
NOWAIT^DEPTH := 1;
ERROR := FILE_OPEN_(TERM1:LENGTH,
                    TERM^NUM1,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...

ERROR := FILE_OPEN_(TERM2:LENGTH,
                    TERM^NUM2,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...
  .
  .
CALL WRITEREADX(TERM^NUM1,
                BUFFER1,WCOUNT1,
                RCOUNT1,TAG1);
IF <> THEN ...

CALL WRITEREADX(TERM^NUM2,
                BUFFER2,WCOUNT2,
                RCOUNT2,TAG2);
IF <> THEN ...
  .
  .
ANY^FILE := -1;
CALL AWAITIOX(ANY^FILE,
              !buffer^address!,
              BYTES,
              TAG);
  .
  .
```

In the skeleton code shown above, the file number associated with the file of the completed I/O operation is returned in ANY^FILE. The program can now use the variable ANY^FILE to identify the active terminal, so it knows how to process the data read by this operation.

Nowait I/O: An Example

The following example shows a complete working program that uses nowait I/O to time out terminal response.

This example enhances the example given at the end of Section 2, Using the File System. Whenever the program prompts the user to enter a value, a timer is started. If the user does not respond to the prompt within five minutes, the user is logged off and the program terminates.

Before prompting the user to select a function, the program checks whether the user is logged on. If not, no prompt is issued and the program terminates. If the user is logged on, then the program prompts the user to select a function.

The user is asked to log on when the program starts. The program uses a global variable LOGGED^ON as a flag to indicate whether the user is logged on. After successfully logging on, LOGGED^ON is set to 1. If the user fails to respond to any prompt within the timeout period, then LOGGED^ON gets set to 0.

As with the example in Section 2, Using the File System, the data file to contain the log must exist before the program is run. You can create this file using either the TACL CREATE command or the FUP CREATE command. It is important that the file you create is named in the appropriate call to the FILE_OPEN_ procedure in the program.

The code for this program appears on the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !maximum file-name
                                         ! length
LITERAL BUFSIZE = 512;
LITERAL WAIT^TIME = 30000D;     !wait up to 5 minutes for
                                 ! input

STRING .SBUFFER[0:BUFSIZE];      !I/O buffer (one extra char)
STRING .S^PTR;                   !pointer to end of string
INT     LOGNUM;                  !log file number
INT     TERMNUM;                 !terminal file number
INT     LOGGED^ON;               !nonzero if someone is
                                 ! logged on

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,AWAITIOX,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_,SETMODE)
?LIST
```

```
!-------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and
! print messages.
!-------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT(N) =

         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print the line:

    DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

    DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

    DEFINE  PRINT^STR(S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is used when the
! file is not open, so there is no file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
   CALL AWAITIOX(TERMNUM);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to do the display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN-1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   CALL AWAITIOX(TERMNUM);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to write messages on the terminal and read the
! user's reply.  If there is an error, this procedure
! attempts to write a message about the error and the program
! is stopped.  If the read takes longer than the defined wait
! time, the procedure returns 1 as its result to signal the
! caller of the timeout.  Otherwise it returns 0.
!------------------------------------------------------------

INT PROC WRITEREADTERM(BUF,LEN,READCOUNT,COUNT^READ);
STRING .EXT BUF;
INT         LEN;
INT         READCOUNT;
INT        .COUNT^READ;
BEGIN
INT         ERR;

!  Prompt the user for input:

   CALL WRITEREADX(TERMNUM,BUF,LEN,READCOUNT);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   CALL AWAITIOX(TERMNUM,
                 !buffer^address!,
                 COUNT^READ,
                 !tag!,
                 WAIT^TIME);
   IF <> THEN

!  Check for timeout:

   BEGIN
      CALL FILE_GETINFO_(TERMNUM,ERR);
      IF ERR = 40 THEN RETURN 1;
      CALL FILE^ERRORS(TERMNUM);
   END;

   BUF[COUNT^READ] := 0;
   RETURN 0;
END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user to log on.  If logon is
! successful, the global variable LOGGED^ON is set to 1.
!-------------------------------------------------------------

PROC LOGON;
BEGIN
   LITERAL NAMESIZE = 20;
   LITERAL PWSIZE = 10;
   STRING .USER^NAME[0:NAMESIZE - 1];
   INT     NAMELEN;
   STRING .PASSWORD[0:PWSIZE - 1];
   INT     PWLEN;
   INT     I;

!  Space down five lines and announce logon:

   FOR I := 1 TO 5 DO PRINT^BLANK;
   PRINT^STR("Please log on");

!  Loop until logon is successful:

   DO
   BEGIN

   !  Request user name:

      PRINT^BLANK;
      SBUFFER ':=' "User name: " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      NAMESIZE);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      CALL AWAITIOX(TERMNUM,
                    !buffer^address!,
                    NAMELEN);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      USER^NAME ':=' SBUFFER FOR NAMELEN;

   !  Request user's password, disabling echo of the input:

      CALL SETMODE(TERMNUM,20,0);
      SBUFFER ':=' "Password: " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      PWSIZE);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      CALL AWAITIOX(TERMNUM,
                    !buffer^address!,
                    PWLEN);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      CALL SETMODE(TERMNUM,20,1);
      PASSWORD ':=' SBUFFER FOR PWLEN;
      !  Perform application-dependent check to verify the
      !  user name and password and set LOGGED^ON to true
      !  if successful.  This example does no checking.

         LOGGED^ON := 1;

      !  Erase the password as soon as it is no longer needed:

         PASSWORD ':=' [PWSIZE * [" "]];

      END UNTIL LOGGED^ON;
   END;


   !------------------------------------------------------------
   ! This procedure asks the user for the next function to do:
   !
   ! "r" to read records
   ! "a" to append a record
   ! "x" to exit the program
   !
   ! The selection made is returned as the result of the call.
   !------------------------------------------------------------

   INT PROC GET^COMMAND;
   BEGIN
      INT    COUNT^READ;

   !  Prompt the user for the function to be performed:

      PRINT^BLANK;
      PRINT^STR("Type 'r' for Read Log, ");
      PRINT^STR("     'a' for Append to Log, ");
      PRINT^STR("     'x' for Exit.  ");
      PRINT^BLANK;

      SBUFFER ':=' "Choice: " -> @S^PTR;
      IF WRITEREADTERM(SBUFFER,@S^PTR '-' @SBUFFER,
                        BUFSIZE,COUNT^READ) THEN RETURN "x";
      RETURN SBUFFER[0];
   END;
```

```
!-------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r." The start of the read is selected randomly by record
! number.  The user has the option of sequentially reading
! subsequent messages.
!-------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT    COUNT^READ;
   INT    ERROR;

!  Position current-record and next-record pointers to the
!  beginning of the file:

   CALL POSITION(LOGNUM, 0D);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

!  Loop reading and displaying records until user declines
!  to read next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Read a record from the log file and display it on the
   !  terminal.  Print "No Such Record" if end of file
   !  reached:

      CALL READX(LOGNUM,SBUFFER,BUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(LOGNUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(LOGNUM);
      END;

      CALL WRITE^LINE(SBUFFER,COUNT^READ);

      PRINT^BLANK;

   !  Prompt the user to read the next record (user must
   !  respond "y" to accept, otherwise return to select
   !  next function):

      SBUFFER ':=' ["Do you want to read another ",
                                "record (y/n)? "]
                    -> @S^PTR;
      IF WRITEREADTERM(SBUFFER,@S^PTR '-' @SBUFFER,
                     BUFSIZE,COUNT^READ) THEN
      BEGIN
         LOGGED^ON := 0;
         RETURN;
      END;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;


!------------------------------------------------------------
! Procedure for appending a record.  The user selected
! function "a." The user is prompted to enter comments.  The
! procedure puts the comments in a new record at the end of
! the file.
!------------------------------------------------------------

PROC APPEND^RECORD;
BEGIN
   INT    COUNT^READ;

   PRINT^BLANK;

!  Prompt user for comments and read comments into the
!  buffer:

   SBUFFER ':=' "Enter today's comments:  "
                  -> @S^PTR;
   IF WRITEREADTERM(SBUFFER,@S^PTR '-' @SBUFFER,
                     BUFSIZE,COUNT^READ) THEN
   BEGIN
      LOGGED^ON := 0;
      RETURN;
   END;

!  Blank out portion of buffer past last character read:

   SBUFFER[COUNT^READ] ':=' " " & SBUFFER[COUNT^READ]
                                FOR BUFSIZE-COUNT^READ BYTES;

!  Place the next-record pointer at the end of file and
!  write the new record there:

   CALL POSITION (LOGNUM,-1D);
   IF <> THEN  CALL FILE^ERRORS(LOGNUM);

   CALL WRITEX(LOGNUM,SBUFFER,BUFSIZE);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "a," or "x."
!------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and then
!  return to prompt again for a valid function:

   PRINT^STR ("INVALID COMMAND: "  &
              "Type either 'r,' 'a,' or 'x'");
END;

!------------------------------------------------------------
! Procedure to initialize the program. It calls
! INITIALIZER to dispose of the startup sequence of messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .LOGNAME[0:MAXFLEN - 1];      !name of log file
   INT     LOGLEN;                      !length of log name
   STRING .TERMNAME[0:MAXFLEN - 1];     !terminal file
   INT     TERMLEN;                     !length of term name
   INT     ERROR;

!  Read and discard startup sequence of messages.

   CALL INITIALIZER;

!  Open the terminal file for nowait I/O.  For simplicity
!  this program uses the home terminal; the recommended
!  approach is to use the IN file read from the Startup
!  message; see Section 8, Communicating With a TACL Process,
!  for details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,
                       TERMNUM,
                       !access!,
                       !exclusion!,
                       1);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the log file with a sync depth of 1:

   LOGNAME ':=' "$ADMIN.OPERATOR.LOGFILE" -> @S^PTR;
   LOGLEN := @S^PTR '-' @LOGNAME;
   ERROR := FILE_OPEN_(LOGNAME:LOGLEN,
                       LOGNUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN, ERROR);

!  Clear the LOGGED^ON flag:

   LOGGED^ON := 0;

END;
```

```
!--------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!--------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely:

   WHILE 1 DO
   BEGIN

   !  Prompt the user to log on:

      CALL LOGON;

   !  Loop until user types "x" or does not answer a prompt:

      WHILE LOGGED^ON DO
      BEGIN

      !  Prompt for the next command.

         CMD := GET^COMMAND;

      !  Call the function selected by user:

         CASE CMD OF
         BEGIN

            "r", "R"  -> CALL READ^RECORD;

            "a", "A"  -> CALL APPEND^RECORD;

            "x", "X"  -> LOGGED^ON := 0;

            OTHERWISE -> CALL INVALID^COMMAND;
         END;
      END;
   END;
END;
```

Using FILE_COMPLETE_ and its Companion Procedures

The FILE_COMPLETE_ procedure and its companion procedures, FILE_COMPLETE_SET_ and FILE_COMPLETE_GETINFO_, provide additional capabilities for programs that use nowait I/O. They combine and enhance the function of the AWAITIO[X] procedures and the Open System Services (OSS) `select()` function. For example, you can use these procedures to complete nowait I/O on OSS files in parallel with nowait I/O on Guardian files. You can define a particular set of files to be “enabled” for completion and then complete outstanding I/O operations on these files over a series of calls.

In brief, these procedures are used as follows:

- The FILE_COMPLETE_SET_ procedure enables a single file or set of files for completion by subsequent calls to the FILE_COMPLETE_ procedure. This set of files can include both Guardian and OSS files.
- The FILE_COMPLETE_GETINFO_ procedure returns information about the set of files that are currently enabled for completion.
- The FILE_COMPLETE_ procedure completes one previously initiated I/O operation for a Guardian file or returns ready information for an OSS file. The Guardian or OSS file is from the set of files that have been enabled for completion.

The following paragraphs explain in more detail how these procedures are used.

Using the FILE_COMPLETE_SET_ Procedure

You can use the FILE_COMPLETE_SET_ procedure to enable a single file or set of files for completion by subsequent calls to the FILE_COMPLETE_ procedure. A file that is “enabled for completion” is part of the set of files that the FILE_COMPLETE_ and FILE_COMPLETE_GETINFO_ procedures are aware of and can act upon; files that are not part of this set are ignored by these procedures. You can add a file to the enabled set only if it has been opened in a nowait manner; OSS files can be opened blocking or nonblocking.

In a call to the FILE_COMPLETE_SET_ procedure, you can add files or remove files from the enabled set. To do this, you pass an array of one or more COMPLETE^ELEMENT structures to the FILE_COMPLETE_SET_ procedure. Each structure describes a file to be added to or removed from the enabled set. Each described file can be either a Guardian file or an OSS file.

The COMPLETE^ELEMENT structure is defined in the ZSYS* files. In the TAL ZSYSTAL file, it is defined as follows:

```
STRUCT ZSYS^DDL^COMPLETE^ELEMENT^DEF (*);
   BEGIN
   INT(32)  Z^FNUM^FD;
   STRUCT   Z^OPTIONS;
      BEGIN
      UNSIGNED(1)  Z^SET^FILE;
      UNSIGNED(1)  Z^FILETYPE;
      BIT_FILLER   14;
      BIT_FILLER   13;
      UNSIGNED(1)  Z^READ^READY;
      UNSIGNED(1)  Z^WRITE^READY;
      UNSIGNED(1)  Z^EXCEPTION;
      END;
   INT(32)  Z^COMPLETION^TYPE = Z^OPTIONS;
   END;
```

By enabling a Guardian file and specifying file number -1D in the Z^FNUM^FD field of this structure, you cause all Guardian files to be enabled for completion (that is, all Guardian files that your program has open for nowait I/O). By removing Guardian file number -1D, you remove all Guardian files from the enabled set. With OSS files, you can specify -1D only to remove all OSS files from the enabled set.

In the call to the FILE_COMPLETE_SET_ procedure, you must also pass a value that represents the total number of files described in the array parameter. Optionally, you can include an output parameter that returns an index to an element in the array parameter if it is in error.

The following example specifies one Guardian file and one OSS file to be added and one OSS file to be removed from the set of enabled files:

```
?SOURCE ZSYSTAL( ZSYS^DDL^COMPLETE^ELEMENT )

LITERAL MAX_COMPLETE_ELEMENTS = 20;

STRUCT .COMPLETE_LIST
 (ZSYS^DDL^COMPLETE^ELEMENT^DEF) [0:MAX_COMPLETE_ELEMENTS -
1];

INT(32) FILENUM;
INT(32) FILEDESC1;
INT(32) FILEDESC2;

INT ERROR_ELEMENT;
INT NUM_ELEMENTS;
INT ERROR;
 .
 .
```

```
-- Describe Guardian file to be added to the enabled set
COMPLETE_LIST[0].Z^FNUM^FD := FILENUM;     -- Guardian file
                                           --  number
COMPLETE_LIST[0].Z^OPTIONS.Z^SET^FILE := 0;-- Add this file
to
                                           --  the enabled
set
COMPLETE_LIST[0].Z^OPTIONS.Z^FILETYPE := 0;-- This is a
                                           --  Guardian file

-- Describe OSS file to be added to the enabled set
COMPLETE_LIST[1].Z^FNUM^FD := FILEDESC1;   -- OSS file
                                           --  descriptor
COMPLETE_LIST[1].Z^OPTIONS.Z^SET^FILE := 0;-- Add this file
to
                                           --  the enabled
set
COMPLETE_LIST[1].Z^OPTIONS.Z^FILETYPE := 1;-- This is an OSS
                                           --  file

-- The following fields are only used when enabling an OSS
file
COMPLETE_LIST[1].Z^OPTIONS.Z^READ^READY := 1; -- Return read
                                              --  ready
COMPLETE_LIST[1].Z^OPTIONS.Z^WRITE^READY := 1;-- Return write
                                              --  ready
COMPLETE_LIST[1].Z^OPTIONS.Z^EXCEPTION := 1;  -- Return
                                              --  exception
                                              --   occurred

-- Describe OSS file to be removed from the enabled set
COMPLETE_LIST[2].Z^FNUM^FD := FILEDESC2;   -- OSS file
                                           --  descriptor
COMPLETE_LIST[2].Z^OPTIONS.Z^SET^FILE := 1;-- Remove this
file
                                           --  from enabled
set
COMPLETE_LIST[2].Z^OPTIONS.Z^FILETYPE := 1;-- This is an OSS
                                           --  file

NUM_ELEMENTS := 3; -- Number of elements (files described)
ERROR := FILE_COMPLETE_SET_ (
                COMPLETE_LIST -- in; element list
               ,NUM_ELEMENTS  -- in; number of elements
               ,ERROR_ELEMENT -- out; index to element in
error
                            );
IF ERROR <> 0 THEN ...
```

Each file that is added to the enabled set remains in the set until your program removes or closes the file. Completion on a file does not remove it from the enabled set.

Using the FILE_COMPLETE_GETINFO_ Procedure

You can use the FILE_COMPLETE_GETINFO_ procedure to obtain information about the set of files that are currently enabled for completion.

Through an output parameter, the FILE_COMPLETE_GETINFO_ procedure returns an array of COMPLETE^ELEMENT structures that describe the files that are enabled for completion. This structure is the same as that used to specify a file to the FILE_COMPLETE_SET_ procedure. (See the structure definition ZSYS^DDL^COMPLETE^ELEMENT^DEF under Using the FILE_COMPLETE_SET_ Procedure.)

When calling the FILE_COMPLETE_GETINFO_ procedure, you must specify the maximum number of structures that your program can accept as output from the procedure. There is also an optional output parameter that returns the actual number of structures that are returned. This number is equal either to the number of files enabled for completion or to the maximum number of COMPLETE^ELEMENT structures that can be returned.

In the following example, a call is made to the FILE_COMPLETE_GETINFO_ procedure:

```
?SOURCE ZSYSTAL( ZSYS^DDL^COMPLETE^ELEMENT )

LITERAL MAX_COMPLETE_ELEMENTS = 20;

STRUCT .COMPLETE_LIST
 (ZSYS^DDL^COMPLETE^ELEMENT^DEF) [0:MAX_COMPLETE_ELEMENTS -
1];

INT NUM_ELEMENTS_OUT;
INT ERROR;
 .
 .
ERROR := FILE_COMPLETE_GETINFO_ (
             COMPLETE_LIST         -- out; set of enabled
files
            ,MAX_COMPLETE_ELEMENTS -- in; max elements
            ,NUM_ELEMENTS_OUT      -- out; number of returned
                                   --  elements
                                );
IF ERROR <> 0 THEN ...
```

Using the FILE_COMPLETE_ Procedure

You can use the FILE_COMPLETE_ procedure to complete one previously initiated I/O operation on a Guardian file or to return ready information on an OSS file. The Guardian file or OSS file is from the set of files that are enabled for completion. You can have the procedure either check for completion and immediately return or wait until either a completion or a timeout occurs.

Completion on Guardian Files

You can use the FILE_COMPLETE_ procedure to complete I/O operations on the same Guardian files as the AWAITIO[X] procedures. (Refer to the discussion of the AWAITIO[X] procedures earlier in this section.) It is possible to use the FILE_COMPLETE_ procedure in parallel with the AWAITIO[X] procedures in your program.

In general, when completing I/O on Guardian files, the FILE_COMPLETE_ procedure behaves very similarly to AWAITIO[X], although one major difference is that it uses a predefined set of files that are enabled for completion. For a list of specific differences between FILE_COMPLETE_ and AWAITIO[X], refer to the description of the FILE_COMPLETE_ procedure in the *Guardian Procedure Calls Reference Manual*.

Completion on OSS Files

Completion on an OSS file means checking for readiness. The file is ready if data can be sent, if data can be received, or if an exception occurred. The operation of checking for readiness is equivalent to calling the OSS `select()` function, except that the FILE_COMPLETE_ procedure returns ready information for only one file at a time. It is also possible to use the FILE_COMPLETE_ procedure in parallel with the OSS `select()` function in your program.

For more information on the OSS `select()` function, refer to the `select(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

Calling the FILE_COMPLETE_ Procedure

The only parameter that must be supplied when you call the FILE_COMPLETE_ procedure is an output parameter that returns completion information for the Guardian file that was completed or the OSS file that is ready. This structure is defined in the ZSYS* files. In the TAL ZSYSTAL file, it is defined as follows:

```
STRUCT ZSYS^DDL^COMPLETION^INFO^DEF (*);
   BEGIN
   INT      Z^FILETYPE;
   INT(32)  Z^ERROR;
   INT(32)  Z^FNUM^FD;
   STRUCT   Z^RETURN^VALUE;
      BEGIN
      BIT_FILLER   15;
      BIT_FILLER   1;
      BIT_FILLER   13;
      UNSIGNED(1)  Z^READ^READY;
      UNSIGNED(1)  Z^WRITE^READY;
      UNSIGNED(1)  Z^EXCEPTION;
      END;
   INT(32)  Z^COMPLETION^TYPE = Z^RETURN^VALUE;
   INT(32)  Z^COUNT^TRANSFERRED = Z^RETURN^VALUE;
   INT(32)  Z^TAG;
   END;
```

For definitions of the fields of this structure, refer to the description of the FILE_COMPLETE_ procedure in the *Guardian Procedure Calls Reference Manual*.

The first optional parameter is the *`timelimit`* parameter, which is used the same way that the *`timeout`* parameter to the AWAITIO[X] procedures, described earlier in this section, is used. However, the following differences exist:

- Error 40, which is returned by the FILE_COMPLETE_ procedure if you specify a *`timelimit`* value other than -1D and an I/O operation times out, does not cause any outstanding I/O operation to be canceled; the operation is considered incomplete.
- Error 26 is returned by the FILE_COMPLETE_ procedure only if you specify a *`timelimit`* value of -1D but no I/O operation has been initiated.

The other optional parameters together provide a means for you to supply a set of files to be temporarily enabled for completion. This set overrides the set of files that were enabled by previous calls to the FILE_COMPLETE_SET_ procedure, but only for the current call to FILE_COMPLETE_. You specify the temporary set of enabled files in much the same manner as the “permanent” set: by supplying an array of COMPLETE^ELEMENT structures that describe the files, except that the array is supplied directly to the FILE_COMPLETE_ procedure. (For a description of the COMPLETE^ELEMENT structure, refer to Using the FILE_COMPLETE_SET_ Procedure.)

Note. For better performance, use the set of files enabled by the FILE_COMPLETE_SET_ procedure rather than specifying a temporary override list to the FILE_COMPLETE_ procedure.

In the following example, the call to the FILE_COMPLETE_ procedure waits for ten seconds for a completion to occur on one of the files in the enabled set; the temporary override set is not used.

```
?SOURCE ZSYSTAL( ZSYS^DDL^COMPLETION^INFO )

STRUCT .COMPLETE_INFO (ZSYS^DDL^COMPLETION^INFO^DEF);
INT(32) TIME_LIMIT;
INT ERROR;
 .
 .
TIME_LIMIT := 1000D; -- Wait for 10 seconds
ERROR := FILE_COMPLETE_ (
                 COMPLETE_INFO_ -- out; info on completed
file
                ,TIME_LIMIT     -- in; time limit on
completion
                        );
IF ERROR <> 0 THEN ...
```

Nowait-Depth

The *nowait-depth* parameter is used by requesters when opening files.The *nowait-depth* parameter tells the file system and the server the maximum number of I/O requests that can be issued concurrently using the same file number.

Nowait-depth value	**Description**
0	Waited I/O. As soon as an I/O is initiated for this file, the process is suspended until the operation is completed.
1	No-waited I/O. Initiating the I/O operation does not suspend the process. I/O operations must be completed using an AWAITIO call. A *`nowait-depth`* value of 1 is the maximum value allowed for $RECEIVE and disk files. If the application needs to issue concurrent I/O on a disk file, open the file several times and issue each no-wait request using a different file number.
> 1 (greater than 1)	Concurrent no-waited I/O. Here, multiple concurrent I/Os are supported over a single file open. To track the different I/Os, the requester must assign each I/O a unique tag that is returned to the requester when the operation is completed. Although the file system supports *`nowait-depth`* value in the range 0 through 15, many devices and processes do not support open requests with a *`nowait-depth`* value greater than 1.

The file system completes multiple requests against the same file in the exact order issued, even if the reply for a request issued *later* physically arrives *before* the reply for a previously issued request. SETMODE 30 enables the file system to complete I/O requests in the order the server replies.

Note that FILE_OPEN_ can be performed no-waited. In this case FILE_OPEN_ returns a file number immediately, which must be used in a later call to AWAITIO (the open does not complete until AWAITIO returns). Use this when opening a process file to ensure that the opened process reads its $RECEIVE messages. If you call FILE_OPEN_ waited, the requester might hang indefinitely if the server misbehaves.

5 Communicating With Disk Files

Data files are typically either NonStop SQL files or files managed by the Enscribe database record manager. This section discusses Enscribe files. For details of NonStop SQL file management, see the NonStop SQL manuals.

In this section, you will learn about each file type supported by the Enscribe software and how to access such files using Guardian procedure calls. Specifically, this section covers the following topics:

- How to use unstructured files
- How to use structured files, including relative files, entry-sequenced files, and key-sequenced files
- How to use partitioned files

For each type of file, this section outlines the common file-system operations: create, open, position, read and write, lock, rename, close, purge, and alter attributes. Emphasis is given to differences among the file types.

This section does not attempt a comprehensive description of Enscribe files; rather, it outlines the major features and programmatic operations affecting Enscribe files and provides examples. Refer to the *Enscribe Programmer's Guide* for complete details.

Many of the programmatic tasks described in this section can also be done interactively using the File Utility Program (FUP); refer to the *File Utility Program (FUP) Reference Manual* for details.

Accessing Enscribe files through the NonStop Transaction Manager/MP (TM/MP) is not discussed here. You should refer to the *NonStop TM/MP Application Programmer's Guide* for details.

Types of Disk Files

The Enscribe database record manager supports structured and unstructured files. Structured files contain data and control information that allows each data record to be uniquely identified. Unstructured files contain only data; access to data is done using a byte address within the file.

Unstructured Files

The Enscribe database record manager imposes no structure on an unstructured file. Any structure that such a file has is imposed by the application.

The application can access arbitrary portions of the file by setting the record pointers to the desired starting byte positions. The current-record pointer addresses the current record, and the next-record pointer addresses the record that physically follows the current record. This type of file is therefore suitable for sequential access. To randomly access records in an unstructured file, you need to set up the pointers with the byte address of the start of the desired record, relative to the beginning of the file.

Unstructured files are suitable for applications that are sequential in nature or where the application itself provides the structure. Files created by the EDIT and TEDIT programs are typical examples of unstructured files.

Structured Files

The Enscribe software supports the following types of structured files:

- Relative files
- Entry-sequenced files
- Key-sequenced files

The following paragraphs describe the major characteristics of these file types and indicate the types of applications that can best take advantage of each file type.

Relative Files

Relative files are made up of logical records. A fixed amount of storage is used for each record, even though the data saved in the record can be a different length. The fixed length, known as the record length, is therefore the maximum length of a relative-file record.

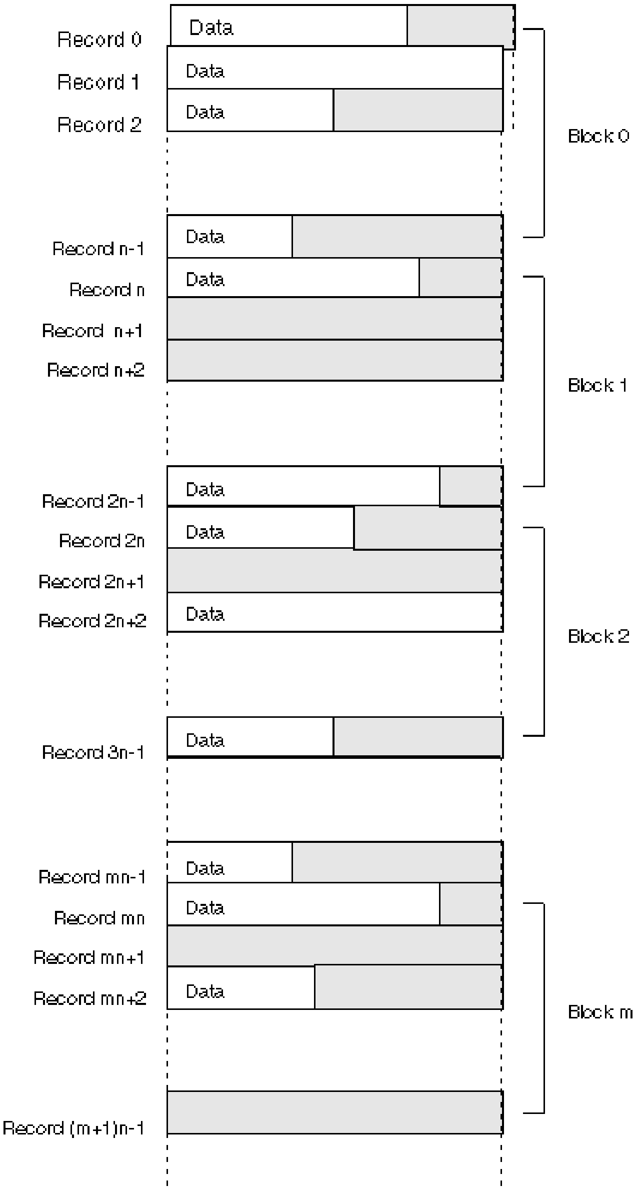
Figure 5-1 shows the structure of a relative file.

Each record in a relative file is identified by a number whose value is the position of the record in the file. The first record in the file is record number 0. You access a given record by setting up the file pointers with the record number and then issuing the appropriate procedure call. Existing records can be updated or deleted. An update can change the record length (up to the maximum length). New records are usually appended to the file.

Relative files are appropriate when you can assign unique numbers to records from a compact range. An employee file indexed on employee number could be a suitable application because the employee number and record number can be the same. If there is a high turnover and employee numbers are never reassigned, however, a key-sequenced file may be more appropriate because the relative file would contain many empty records. A relative file also makes it impossible to include alphabetic characters in the employee ID.

An inventory file indexed by part number might seem to be an appropriate application of relative files. However, part-number schemes do tend to have large gaps; a record would be reserved for each number, whether there was a corresponding part for it or not.

Figure 5-1. Relative-File Structure


Record 0
Data
Record 1
Data
Record 2
Data
Block 0
Record n-1
Data
Record n
Data
Record n+1
Record n+2
Block 1
Record 2n-1
Data
Record 2n
Data
Record 2n+1
Record 2n+2
Data
Block 2
Record 3n-1
Data
Record mn-1
Data
Record mn
Data
Record mn+1
Record mn+2
Data
Block m
Record (m+1)n-1

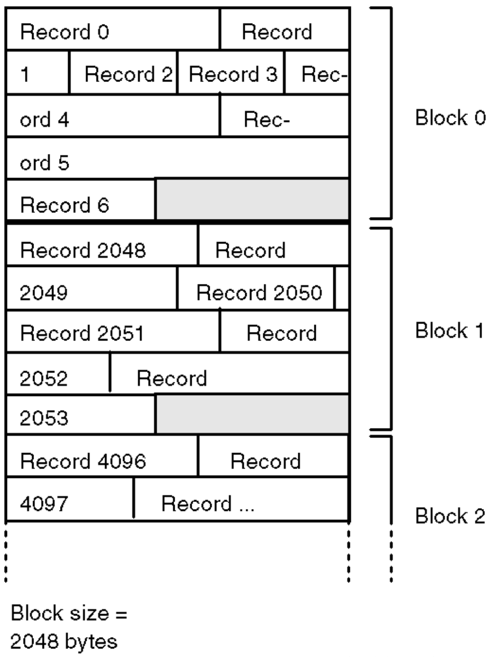
VST023.VSD

Entry-Sequenced Files

Records in an entry-sequenced file are variable length. Writing to the file involves appending records to the end of the file. Records therefore appear in the file in the order in which they were written. Once the record is written, its size cannot be changed.

Figure 5-2 shows the structure of an entry-sequenced file.

Figure 5-2. Entry-Sequenced File Structure


Record 0
Record
1
Record 2
Record 3
Rec-
ord 4
Rec-
ord 5
Record 6
Record 2048
Record
2049
Record 2050
Record 2051
Record
2052
Record
2053
Record 4096
Record
4097
Record ...
Block 0
Block 1
Block 2
Block size =
2048 bytes

VST024.VSD

The key to an entry-sequenced file is the record address, made up of the block byte address and the record number within that block. You use the record address to access a given record. Because you cannot change the length of a record in an entry-sequenced file, records are usually not updated, unless you do so without changing the record length.

Entry-sequenced files are useful when records of variable length are anticipated in an application that stores data chronologically. A transaction logging file, for example, saves a record of information for each transaction in the order in which the transactions occurred.

Key-Sequenced Files

In key-sequenced files, each record is identified by a unique key that is stored in the record itself. With key-sequenced files, you can read records, insert new records, delete records, and update records. When updating records, you also have the possibility of changing the length of the record.

Tree-structured index records provide random access to the data records. Data records can also be accessed sequentially in key sequence.

Key-sequenced files are “tree structured.” The trunk of the tree is an index block containing records that each point to a second level of index block. The second-level index blocks contain pointers to a third level, and so on. Finally, the lowest level of index block contains pointers to the leaves of the tree that contain the data records.

Figure 5-3 shows the structure of a key-sequenced file. This example shows two levels of index blocks. The second level of index blocks points directly to the data blocks.

Note. All index blocks and data blocks of a key-sequenced file reside in the same file.

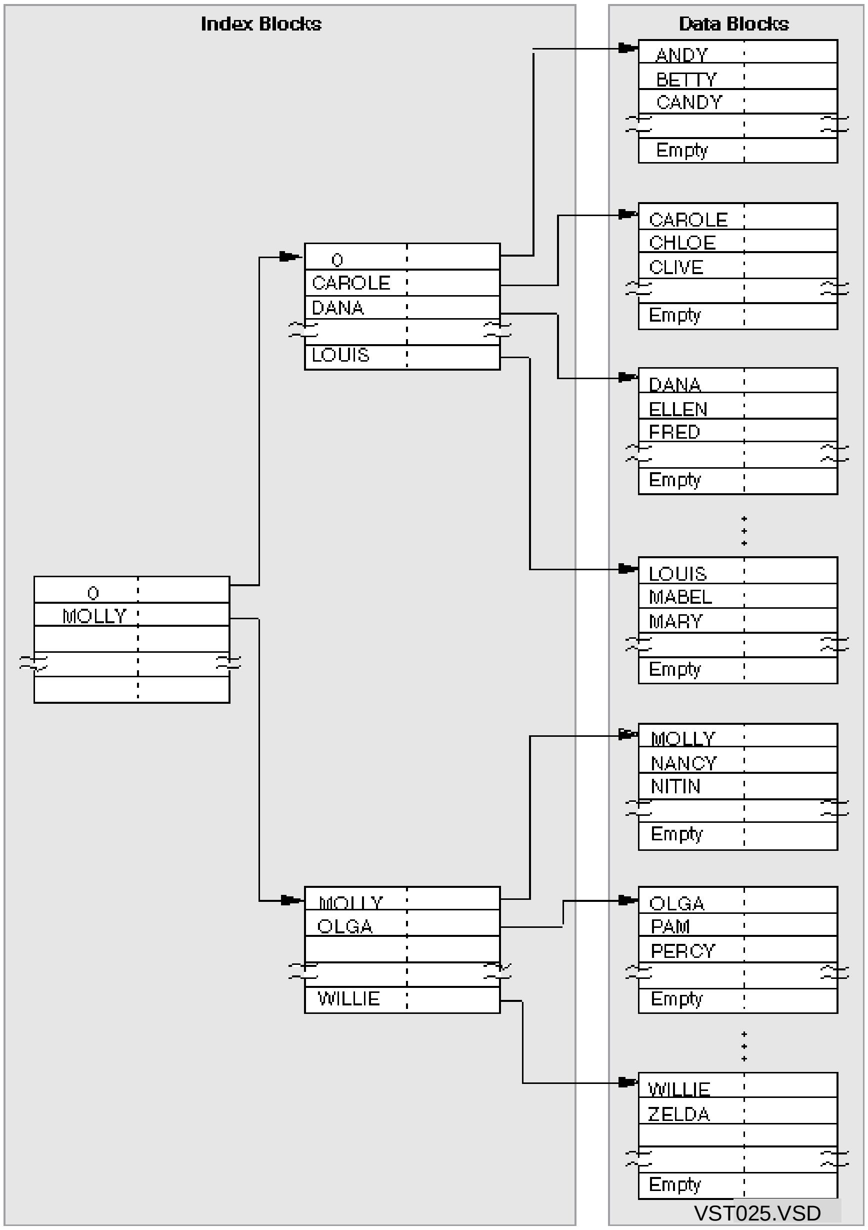
You access data records randomly by specifying the key value of a record. The search starts by comparing the supplied key with the record keys in the highest-level index block. The system software finds the highest key in the block that is less than or equal to the supplied key value. The corresponding record contains a pointer to a second-level index block where the key comparison is repeated. By traversing index blocks in this way, the Enscribe software finally arrives at the data block that contains the desired record. A search of this block locates the record.

When adding a record to a key-sequenced file, the data record is added to the file and the index records are updated accordingly. Key-sequenced files are initially set up with empty records in the data blocks to enable records to be added efficiently. When a data block is full, the Enscribe software creates another and sets up the index pointers accordingly.

Key-sequenced files also support sequential access of data records by key sequence.

Key-sequenced files are suitable for any application where random access by key value is required. An inventory file where each record describes a part could be set up as a key-sequenced file, using the part number as the unique key. A banking system organized by account number is another typical example.

Figure 5-3. Key-Sequenced File Structure


Index Blocks
Data Blocks
0
MOLLY
0
CAROLE
DANA
LOUIS
MOLLY
OLGA
WILLIE
ANDY
BETTY
CANDY
Empty
CAROLE
CHLOE
CLIVE
Empty
DANA
ELLEN
FRED
Empty
LOUIS
MABEL
MARY
Empty
MOLLY
NANCY
NITIN
Empty
OLGA
PAM
PERCY
Empty
WILLIE
ZELDA
Empty
VST025.VSD

Alternate-Key Files

An alternate-key file is a key-sequenced file that has a special association with a primary file. The primary file can be a relative file, an entry-sequenced file, or another key-sequenced file. This association allows the file system to use the records in the alternate-key file to keep track of records in the primary file.

An alternate-key file can be opened and read or updated by itself, just like any key-sequenced file.

Although the main purpose of an alternate-key file is to provide alternate access to records in primary files, the ability to manipulate the alternate file on its own is sometimes useful. For example, when all data for some application function is contained within the alternate-key file and the primary file is large, there can be significant performance advantage to accessing the smaller alternate-key file on its own.

A record in any structured file can be accessed by one or more alternate keys. For example, it might be useful for a banking application to be able to access an account by name as well as account number. Alternate-key files provide the access mechanism.

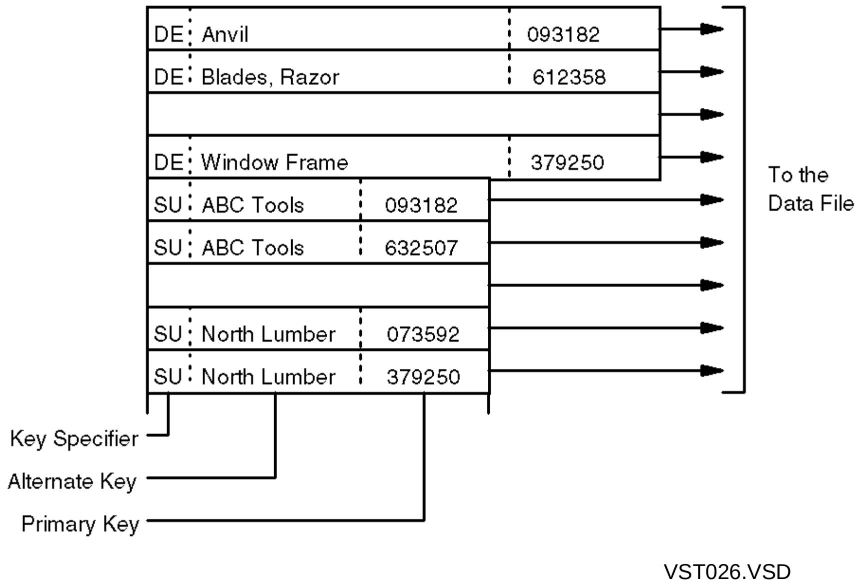
An alternate-key file contains a record for each valid alternate key. This record contains three fields:

- The alternate-key value.
- A key specifier: a two-byte value that distinguishes among different alternate keys in the same alternate-key file. For example, an inventory application might use part description and supplier name as secondary keys; the key-specifier field indicates whether a given record uses a part description or a supplier name as the secondary key.
- The primary key of the corresponding data record. For a relative file, the primary key is the record number. For an entry-sequenced file, the primary key is the record address. For a key-sequenced file, the primary key is the key value embedded in the record itself.

For alternate-key files containing nonunique alternate-key values, records with like key specifiers are always contained in the same alternate-key file. An alternate-key file may contain more than one key specifier. In other words, all alternate keys can be contained in the same alternate-key file, or they can be segregated according to key type. Figure 5-4 shows several key specifiers in the same alternate-key file.

Records with unique alternate keys are always contained in the same file. They are never kept in the same file with keys that are a different length.

Figure 5-4. An Alternate-Key File


DE
Anvil
093182
DE
Blades, Razor
612358
DE
Window Frame
379250
SU
ABC Tools
093182
SU
ABC Tools
632507
SU
North Lumber
073592
SU
North Lumber
379250
To the
Data File
Key Specifier
Alternate Key
Primary Key
VST026.VSD

When a new key is added to the data file, new alternate keys are automatically added to the alternate-key file (unless the alternate key is defined as not automatically updated).

Unlike primary keys, alternate keys can be duplicated if the file is designated to accept duplicate keys. Duplicate keys are added to the alternate-key file in the same order as the corresponding key in the primary file. Figure 5-4 shows examples of duplicate keys.

Duplicate alternate keys can, for example, provide keyed access for two people with the same name; their primary keys would, of course, have unique values such as a bank account number or social security number.

Queue Files

A queue file is a special type of key-sequenced disk file that can function as a queue. Processes can queue and dequeue records in a queue file.

Queue files contain variable-length records that are accessed by values in designated key fields. Unlike other key-sequenced files, queue files have primary keys but cannot have alternate keys. The primary key for a queue file includes an eight-byte timestamp; you can add a user key if desired. The disk process inserts the timestamp when each record is inserted into the file, and maintains the timestamp during subsequent file operations.

For more information about queue files and how to use them, refer to the *Enscribe Programmer’s Guide*.

Using Unstructured Files

You can access unstructured files using system procedures such as FILE_OPEN_, READ[X], READUPDATE[X], WRITE[X], WRITEUPDATE[X], and so on. Unstructured files are suitable for sequential I/O; successive calls to the READX procedure, for example, read successive records from the file. Positioning is done by the file system, which advances the current-record and next-record pointers as records are read. Much of the method for working with unstructured files has already been discussed in Section 2, Using the File System. This subsection emphasizes how to create unstructured files.

The IOEdit subset of procedures, provides additional functions for accessing unstructured EDIT files. Refer to Section 14, Using the IOEdit Procedures, for details.

As an aid to sequential-file access, the procedure library contains another subset of procedures specifically for sequential I/O. These procedures are known as the sequential input/output (SIO) procedures and are described in Section 15, Using the Sequential Input/Output Procedures.

For an example of accessing an unstructured file using READX, WRITEX, and POSITION procedure calls, refer to the log-file program given at the end of Section 2, Using the File System.

Creating Unstructured Files

You can create an unstructured file interactively using the FUP CREATE command or programmatically by issuing a call to the FILE_CREATE[LIST]_ procedure. In either case, you need to supply the following information:

- The file type for an unstructured file. Unstructured is the default file type.
- The size of the buffer used to transfer data between the disk and the disk process for an unstructured file. The buffer size can be 512, 1024, 2048, or 4096 bytes. The default block size is 4096 bytes.

  You can improve the efficiency of the disk cache management scheme by setting the buffer size to the same size as each data transfer. See the discussion on transfer size in the *Enscribe Programmer’s Guide* for details.

The following example interactively creates an unstructured file with a buffer size of 512 bytes using the FUP CREATE command:

```
1> FUP
-SET TYPE U
-SET BUFFERSIZE 512
-SHOW
    TYPE U
    EXT ( 1 PAGES, 1 PAGES )
    MAXEXTENTS 16
    BUFFERSIZE 512
-CREATE $ADMIN.OPERATOR.LOGFILE
CREATED - $ADMIN.OPERATOR.LOGFILE
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

The following example programmatically creates the same file using the FILE_CREATE_ procedure:

```
STRING .FILE^NAME[0:ZSYS^VAL^LEN^FILENAME - 1];
INT LENGTH;
INT FILE^TYPE := 0;
INT BUFFER^SIZE := 512;
 .
 .
FILE^NAME ':=' "\SYS.$ADMIN.OPERATOR.LOGFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
CALL FILE_CREATE_(FILE^NAME:ZSYS^VAL^LEN^FILENAME,
                  LENGTH,
                  !file^code!,
                  !primary^extent^size!,
                  !secondary^extent^size!,
                  !max^extents!,
                  FILE^TYPE,
                  !options!,
                  !record^len!,
                  BUFFER^SIZE);
```

As for any disk file, you should also consider the desired extent sizes for your file. The above example creates a file using the default extent sizes of 1 page for the primary extent and 1 page for each of the secondary extents. Refer to Section 2, Using the File System, for details of setting extent sizes.

Opening Unstructured Files

You open an unstructured file as you would any other file, by using the FILE_OPEN_ procedure.

```
INT FILE^NUM;
 .
 .
CALL FILE_OPEN_(FILE^NAME:LENGTH,
                FILE^NUM,
                !access!,
                !exclusion!,
                !nowait^depth!,
                SYNC^DEPTH);
```

This example opens the file for reading and writing with waited I/O. Refer to Section 3, Coordinating Concurrent File Access, for information on access and exclusion modes and to Section 4, Using Nowait Input/Output, for a discussion of waited versus nowait I/O.

A sync depth of 1 permits retryable write requests against the file when automatically recovering from path errors. In other words, the sync depth ensures that your data does not get corrupted due to path errors.

Positioning, Reading, and Writing With Unstructured Files

Unstructured files can be accessed sequentially. Random access is also possible if the byte address of the data you need is known; you set the byte address using the POSITION procedure. To begin sequential access anywhere except at the beginning of the file, you need to set the pointers to the starting byte address.

Refer to Section 2, Using the File System, for details of how to access data in an unstructured file, including a discussion on the function of the file pointers.

Locking With Unstructured Files

Sometimes you need to ensure exclusive access to a given file or record for a limited time, for example, while a transaction is in progress. As with any disk file, you can lock other processes out of an unstructured file by using the LOCKFILE procedure or out of a given record by using the LOCREC procedure.

Section 3, Coordinating Concurrent File Access, describes how to use the LOCKFILE procedure to acquire a file lock, and how to remove a file lock using the UNLOCKFILE procedure. LOCKREC and UNLOCKREC work in a similar way, as follows:

```
CALL POSITION(FILE^NUM,
              RECORD^ADDRESS);
CALL LOCKREC(FILE^NUM);
IF <> THEN ...                   !could not get the lock
 .
 .
!protected I/O operations
 .
 .
CALL UNLOCKREC(FILE^NUM);
```

The LOCKREC procedure locks the current record (the one addressed by the current-record pointer). UNLOCKREC removes the lock from the current record. Use care to ensure that the file pointers are positioned correctly when unlocking the record. If you have used sequential reads or writes or have done multiple I/O operations to the file since locking the record, you will need to reset the pointers before unlocking the record.

Note. For files that are protected by the NonStop Transaction Manager/MP (TM/MP), the acquisition and release of record locks is different from using the Enscribe procedures without NonStop TM/MP protection. Every modified record gets an implicit lock that is not released, even by explicit unlock requests, until the transaction ends. Refer to the *NonStop TM/MP Application Programmer's Guide* for details.

Renaming Unstructured Files

You rename an unstructured file as you would any other file, by using the FILE_RENAME_ procedure. The following procedure call renames the file opened with file number FILE^NUM.

```
NAME ':=' "\SYS.$ADMIN.OPERATOR.LOGFILE1" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @NAME;
ERROR := FILE_RENAME_(FILE^NUM,
                      NAME:NAME^LENGTH);
IF ERROR <> 0 THEN ...
```

Note. You can rename only the subvolume and file ID parts of the file name. You cannot change the name of the volume on which the file resides. The volume specified must be the same as the volume the file already resides on.

Avoiding Unnecessary Cache Flushes to Unstructured Files

You can avoid unnecessary cache flushes to unstructured files in the same way as for any other file by using function 152 of the SETMODE procedure as shown in the following code fragment:

```
LITERAL AVOID^FLUSH = 152,
        DONT^FLUSH  = 1;
  .
  .
CALL SETMODE(FILE^NUM,
             AVOID^FLUSH,
             DONT^FLUSH);
IF <> THEN ...
```

By default, an unaudited file always has its cache flushed when you close the file. By using SETMODE 152, you avoid this unnecessary overhead in the following situations:

- After the close, the file remains open for writing by your process or some other process.
- The close is not the last close on a file that was opened with a nonzero sync-depth value.

To be effective, SETMODE 152 with *`param1`* set to 1 should be performed for each open of the file. Any open that does not perform SETMODE 152 with *`param1`* set to 1 causes a cache flush when it closes the file.

For optimal performance, you should not use SETMODE 152 with *`param1`* set to 1 if the file is to be closed by all openers at about the same time. This is because, in such a case, all buffers in the cache are flushed serially by the last opener rather than in parallel by each opener.

For audited files, SETMODE 152 is unnecessary. The NonStop TM/MP product automatically avoids unnecessary flushes. See the *NonStop TM/MP Reference Manual* for details.

Closing Unstructured Files

An unstructured file can be closed in the same way as any other file, by using the FILE_CLOSE_ procedure:

```
ERROR := FILE_CLOSE_(FILE^NUM);
IF ERROR <> 0 THEN ...
```

When a process terminates, any files that the process still has open are automatically closed.

Purging Unstructured Files

You purge an unstructured file the same way you would purge any file; either interactively using the FUP PURGE command or TACL PURGE command, or programmatically by calling the FILE_PURGE_ procedure. Purging does not normally delete the data, but it changes pointers to show the file to be absent and its extent space deallocated.

The file should be closed before you attempt to purge it.

You can force the FILE_PURGE_ procedure to clear the file of all data by setting the CLEARONPURGE flag. You may want to do this for security reasons; otherwise the extents that the purged file occupied do become readable when reallocated to another file. Use function 1 of the SETMODE procedure to set the CLEARONPURGE flag (bit 1) to 1.

The following example purges a file and clears all its data:

```
LITERAL SET^SECURITY   = 1,
        CLEAR^ON^PURGE = %40000;
INT     OLD^VALUES[0:1];
 .
 .
!Save current security flag values

CALL SETMODE(FILE^NUM,
             SET^SECURITY,
             !param^1!,
             !param^2!,
             OLD^VALUES);

!Set CLEARONPURGE and merge with existing security flag
!values:

CALL SETMODE(FILE^NUM,
             SET^SECURITY,
             (CLEAR^ON^PURGE LOR OLD^VALUES[0]));
IF <> THEN ...
 .
 .
ERROR := FILE^CLOSE(FILE^NUM);
IF ERROR <> THEN ...
 .
 .
NAME ':=' "\SYS.$ADMIN.OPERTOR.LOGFILE1" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @NAME;
ERROR := FILE_PURGE_(NAME:NAME^LENGTH);
IF ERROR <> 0 THEN ...
```

Common reasons for an error to be returned are that some other process has the file open, the owner does not permit this process to purge the file, or the file is a TMF audit file. The Transaction Management Facility subsystem (TMF) is the main functional component of NonStop TM/MP.

Altering Unstructured-File Attributes

As for any other file type, file attributes for an unstructured file are normally set when the file is created. These attributes include, for example, an application-supplied file code or an expiration time before which the file cannot be purged. You can, however, change some attributes of an existing file by calling the FILE_ALTERLIST_ procedure.

Note. When you call FILE_ALTERLIST_ for a given file, the file must **not** be open otherwise the procedure returns an error.

The following example changes the file code for the file named MYFILE and sets TMF auditing for the file. Here, a change of file code is requested by the *`item-list`* parameter (code 42), and a change in TMF audit status by *`item-list`* code 66. The new values are provided in the first and second items listed in the *`values`* parameter.

```
FILENAME ':=' "MYFILE";
NAME^LENGTH := 6;
ITEM^LIST ':=' [42,66];
NUMBER^OF^ITEMS := 2;
VALUES ':=' [125,1];
VALUES^LENGTH := 4;
ERROR := FILE_ALTERLIST_(FILENAME:NAME^LENGTH,
                         ITEM^LIST,
                         NUMBER^OF^ITEMS,
                         VALUES,
                         VALUES^LENGTH);
```

An alternate way of altering file attributes is to use the FUP ALTER command. This command allows you to set attributes interactively through the TACL program instead of programmatically using the FILE_ALTERLIST_ procedure.

See the *Guardian Procedure Calls Reference Manual* for complete details of every file attribute that you can change with the FILE_ALTERLIST_ procedure. See the *Guardian Procedure Calls Reference Manual* for details of the FUP ALTER command.

Using Relative Files

This subsection discusses how to create and access relative files. It outlines the common file-system operations: create, open, position, read and write, lock, rename, close, purge, and alter attributes. The discussion includes a complete program that makes use of the major features of relative files, including the ability to randomly access a file using the record number.

The discussion here is limited to primary-key access, that is, access by record number. Relative files can also be accessed by alternate keys; for details, refer to Using Alternate Keys later in this section.

Creating Relative Files

You can create a relative file either interactively using the FUP CREATE command or programmatically using the FILE_CREATE[LIST]_ procedure. In either case, you need to supply information about how to build the file, including the appropriate file type, block size, and maximum record length.

The following example creates a relative file interactively using the FUP CREATE command:

```
1> FUP
-SET TYPE R
-SET BLOCK 4096
-SET REC 128
-SHOW
    TYPE R
    EXT ( 1 PAGES, 1 PAGES )
    REC 128
    BLOCK  4096
    MAXEXTENTS 16
-CREATE $ADMIN.OPERATOR.RELFILE
CREATED - $ADMIN.OPERATOR.RELFILE
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

The next example creates the same file programmatically using the FILE_CREATE_ procedure:

```
STRING .FILE^NAME[0:ZSYS^VAL^LEN^FILENAME - 1];
INT     LENGTH;
INT     FILE^TYPE := 1;
INT     RECORD^LENGTH := 128;
INT     BLOCK^LENGTH := 4096;
 .
 .
FILE^NAME ':=' "\SYS.$HR.RECORDS.EMPFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_CREATE_(FILE^NAME:ZSYS^VAL^LEN^FILENAME,
                      LENGTH,
                      !file^code!,
                      !primary^extent^size!,
                      !secondary^extent^size!,
                      !max^extents!,
                      FILE^TYPE,
                      !options!,
                      RECORD^LENGTH,
                      BLOCK^LENGTH);
IF ERROR <> 0 THEN ...
```

A file type of 1 specifies a relative file.

The record length is set to 128 bytes, which sets the limit on the size of a logical record. (Recall that the maximum size of a logical record for a relative file is the record length.) A block size of 4096 limits the maximum size of a record that could be specified.

The block size is the number of bytes that are transferred between the disk and the disk process. The block size can be 512, 1024, 2048, or 4096 bytes. Records cannot span blocks; therefore the block size must be at least large enough to contain one record and the overhead associated with the block. In other words, the maximum record size is smaller than the block size. A block usually contains multiple records.

As for any disk file, the file system allocates a primary extent to the file and as many secondary extents as necessary (up to the maximum allowed). The above example assumes the default extent sizes and default maximum number of extents. Extent allocation is described in Section 2, Using the File System.

Opening Relative Files

A relative file is opened in the same way as any other file, by using the FILE_OPEN_ procedure. See Using Unstructured Files, earlier in this section, for details.

Positioning, Reading, and Writing With Relative Files

Before performing a read or write operation on a relative file, you must be sure that the current-record and next-record pointers point to the appropriate places. When you open the file, the pointers are set up to access record 0. If you want to randomly access other records in the file, you must move the pointers using the POSITION procedure to do this. For example:

```
INT(32) RECORD^NUM;
 .
 .
RECORD^NUM := 24D;
CALL POSITION(FILE^NUM,RECORD^NUM);
IF <> THEN ...
```

The above example places the current-record and next-record pointers at the start of record number 24 in the file. Your program can now do sequential read and write operations using the READ[X] or WRITE[X] procedures starting at record number 24. READUPDATE[X] and WRITEUPDATE[X] can be used if you do not wish to move the file pointers, for example, when updating a record.

You can position the file pointers to the next empty physical record by setting the record number to -2D. You can address the end of the file (for appending a record) by setting the record number to -1D.

Locking, Renaming, Caching, Closing, Purging, and Altering Relative Files

The operations of locking, renaming, closing, and purging relative files, altering relative-file attributes, and avoiding unnecessary cache flushes of relative files are the same as for any disk file. See Using Unstructured Filesearlier in this section.

Relative-File Programming Example

This example is an extension of the log-file program described near the end of Section 2, Using the File System. It is modified to use a relative file instead of an unstructured file. A relative file is suitable for this kind of application because:

- Entries in the file are chronological and therefore suitable for referencing by record number.
- The record number gives the user a key to access records randomly.

You can create the relative file required by this program using the FILE_CREATE_ procedure as described under Creating Relative Files, or you can simply use FUP commands as shown below:

```
1> FUP
-SET TYPE R
-SET BLOCK 4096
-SET REC 512
-SHOW
    TYPE R
    EXT ( 1 PAGES, 1 PAGES )
    REC 512
    BLOCK  4096
    MAXEXTENTS 16
-CREATE $ADMIN.OPERATOR.RELFILE
CREATED - $ADMIN.OPERATOR.RELFILE
-EXIT
2>
```

The record length is set by a FUP command to 512 bytes. Each record's data can therefore be any length up to 512 bytes.

In addition to code modified to use a relative file instead of an unstructured file, this program contains a function that updates an existing record.

To make the code use a relative file instead of an unstructured file, the code has been modified in the following ways:

- The UPDATE^RECORD procedure has been added. This procedure allows the user to replace an existing record. It prompts the user for the record number to update, then prompts for the new comments and writes the new contents over the original contents in the file. Finally, the procedure returns control to the LOGGER procedure.

- The READ^RECORD procedure prompts the user for the record number of the first record to be read instead of always starting with the first record in the file.
- The LOGGER and GET^COMMAND procedures support the “u” option for updating a record.

The code for this program appears on the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !maximum file-name
                                         ! length
LITERAL BUFSIZE = 512;

STRING .SBUFFER[0:BUFSIZE];      !I/O buffer (one extra char)
STRING .S^PTR;                   !pointer to end of string
INT     LOGNUM;                  !log file number
INT     TERMNUM;                 !terminal file number

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN)
?LIST
```

```
!--------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and print
! messages.
!--------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT(N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER, 0) #;


!  Print a string:

    DEFINE  PRINT^STR(S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there isn't a file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to do the display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN - 1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!-------------------------------------------------------------
! This procedure asks the user for the next function to do:
!
! "r" to read records
! "u" to update a record
! "a" to append a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'u' for Update Log, ");
   PRINT^STR("     'a' for Append to Log, ");
   PRINT^STR("     'x' for Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r."  The start of the read is selected randomly by record
! number.  The user has the option of sequentially reading
! subsequent messages.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT         COUNT^READ;
   INT(32)     RECORD^NUM;
   STRING .EXT NEXT^ADR;
   INT         STATUS;
   INT         ERROR;

!  Prompt the user to select a record:

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Record Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
   SBUFFER[COUNT^READ] := 0;

!  Convert ASCII to numeric:

   @NEXT^ADR := DNUMIN(SBUFFER,RECORD^NUM,10,STATUS);
   IF STATUS OR @NEXT^ADR <> $XADR(SBUFFER[COUNT^READ]) THEN
   BEGIN
      PRINT^STR("Error in the record number");
      GOTO PROMPT^AGAIN;
   END;

!  Position current-record and next-record pointers to
!  selected record:

   CALL POSITION(LOGNUM,RECORD^NUM);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);
```

```
!  Loop reading and displaying records until user declines
!  to read next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

      !  Read a record from the log file and display
      !  it on the terminal.  If end-of-file is reached,
      !  return control to LOGGER procedure:

      CALL READX(LOGNUM,SBUFFER,BUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(LOGNUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(LOGNUM);
      END;

      CALL WRITE^LINE(SBUFFER,COUNT^READ);

      PRINT^BLANK;

      !  Prompt the user to read the next record (user
      !  must respond "y" to accept, otherwise return
      !  to select next function):

      SBUFFER ':=' ["Do you want to read another ",
                               "record (y/n)? "]
                     -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

      SBUFFER[COUNT^READ] := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

```
!------------------------------------------------------------
! Procedure for updating a record.  The user selected
! function "u."  The user is prompted for the record number
! to update.  The procedure displays the current contents and
! prompts for the new.  After the user enters the new
! contents, the procedure updates the log file.
!------------------------------------------------------------

PROC UPDATE^RECORD;
BEGIN
   INT         COUNT^READ;
   INT(32)     RECORD^NUM;
   STRING .EXT NEXT^ADR;
   INT         STATUS;
   INT         ERROR;

!  Prompt the user to select a record:

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Record Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   SBUFFER[COUNT^READ] := 0;

!  Convert ASCII to numeric:

   @NEXT^ADR := DNUMIN(SBUFFER,RECORD^NUM,10,STATUS);
   IF STATUS OR @NEXT^ADR <> $XADR(SBUFFER[COUNT^READ]) THEN
   BEGIN
      PRINT^STR("Error in the record number");
      GOTO PROMPT^AGAIN;
   END;

!  Position current-record and next-record pointers to
!  selected record:

   CALL POSITION(LOGNUM,RECORD^NUM);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);


!  Read the record without moving the current-record and
!  next-record pointers.  If end-of-file is reported,
!  return to LOGGER:

   CALL READUPDATEX(LOGNUM,SBUFFER,BUFSIZE,COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(LOGNUM,ERROR);
      IF ERROR = 1 THEN
      BEGIN
         PRINT^STR("No such record");
         RETURN;
      END;
```

```
         CALL FILE^ERRORS(LOGNUM);
      END;

   !  Write the record to the terminal screen:

      PRINT^BLANK;
      CALL WRITE^LINE(SBUFFER,COUNT^READ);

   !  Prompt the user for the updated record:

      PRINT^BLANK;
      SBUFFER ':=' "Enter New Contents of Record:  "  -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

   !  Write new record to log file:

      CALL WRITEUPDATEX(LOGNUM,SBUFFER,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(LOGNUM);

   END;


   !------------------------------------------------------------
   ! Procedure for appending a record.  The user selected
   ! function "a."  The user is prompted to enter comments.  The
   ! procedure puts the comments in a new record at the end of
   ! the file.
   !------------------------------------------------------------

   PROC APPEND^RECORD;
   BEGIN
      INT    COUNT^READ;

      PRINT^BLANK;

   !  Prompt user for comments and read comments into the
   !  buffer:

      SBUFFER ':=' "Enter today's comments:  "
                    -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

   !  Place the next-record pointer at the end-of-file and
   !  write the new record there:

      CALL POSITION(LOGNUM, -1D);
      IF <> THEN  CALL FILE^ERRORS(LOGNUM);

      CALL WRITEX(LOGNUM,SBUFFER,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(LOGNUM);
   END;
```

```
!--------------------------------------------------------------
! Procedure to exit the program.
!--------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;

!--------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "u", "a," or "x."
!--------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that his selection was invalid
!  then return to prompt again for a valid function:

   PRINT^STR("INVALID COMMAND: "  &
             "Type either 'r,' 'u,' 'a,' or 'x'");
END;
```

```
!-------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .LOGNAME[0:MAXFLEN - 1];      !name of log file
   INT     LOGLEN;                      !length of log name
   STRING .TERMNAME[0:MAXFLEN - 1];     !terminal file
   INT     TERMLEN;                     !length of term name
   INT     ERROR;

!  Read and discard startup messages:

   CALL INITIALIZER;

!  Open the terminal file.  For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  read from the Startup message; see Section 8, “Communicating
!  With a TACL Process,” for details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the log file with a sync depth of 1:

   LOGNAME ':=' "$ADMIN.OPERATOR.RELFILE" -> @S^PTR;
   LOGLEN := @S^PTR '-' @LOGNAME;
   ERROR := FILE_OPEN_(LOGNAME:LOGLEN,
                       LOGNUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function "x":

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command:

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "u", "U"  -> CALL UPDATE^RECORD;

         "a", "A"  -> CALL APPEND^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
   END;
END;
```

Using Entry-Sequenced Files

Entry-sequenced files have a different structure than relative files, therefore file creation is different. File access is also different; random file access, for example, is done by record address instead of record number. This subsection discusses how to use entry-sequenced files, placing emphasis on file creation and file access because these are the operations that differ from operations performed on other structured file types. At the end of the subsection is a sample program showing I/O operations on an entry-sequenced file.

The discussion here is limited to primary-key access, that is, by record address. Alternate keys are discussed later under the heading Using Alternate Keys.

Creating Entry-Sequenced Files

You can create an entry-sequenced file either interactively using the FUP CREATE command or programmatically using the FILE_CREATE[LIST]_ procedure. In either case, you need to supply information about how to build the file, including the appropriate file type, block length, and record length.

The following example creates an entry-sequenced file interactively using the FUP CREATE command:

```
1> FUP
-SET TYPE E
-SET BLOCK 4096
-SET REC 4072
-SHOW
    TYPE E
    EXT ( 1 PAGES, 1 PAGES )
    REC 4072
    BLOCK  4096
    MAXEXTENTS 16
-CREATE $ADMIN.OPERATOR.ESFILE
CREATED - $ADMIN.OPERATOR.ESFILE
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

The next example creates the same file programmatically using the FILE_CREATE_ procedure:

```
STRING .FILE^NAME[0:ZSYS^VAL^LEN^FILENAME - 1];
INT     LENGTH;
INT     FILE^TYPE := 2;
INT     RECORD^LENGTH := 4072;
INT     BLOCK^LENGTH := 4096;
  .
  .
```

```
FILE^NAME ':=' "\SYS.$HR.RECORDS.ESFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
CALL FILE_CREATE_(FILE^NAME:ZSYS^VAL^LEN^FILENAME,
                  LENGTH,
                  !file^code!,
                  !primary^extent^size!,
                  !secondary^extent^size!,
                  !max^extents!,
                  FILE^TYPE,
                  !options!,
                  RECORD^LENGTH,
                  BLOCK^LENGTH);
```

A file type of 2 specifies an entry-sequenced file.

The maximum record length has been set to 4072 bytes, almost equal to the block size. (Entry-sequenced files require a few bytes of overhead in each block.) There is no need to make the record size any smaller, because space is not wasted when you use records that are smaller than the maximum (unlike relative files, where disk space equal to the maximum record size is allocated even if the record itself is only one byte long). Records can be any length from one byte up to this maximum. Unlike relative files, records follow each other immediately, regardless of their sizes.

In this example, the block size is 4096 bytes. The file system will pack as many records into this block size as it can, then start another block.

Opening Entry-Sequenced Files

Once an entry-sequenced file is created, you can open it as you would any file, by using the FILE_OPEN_ procedure. See Using Unstructured Files for an example of opening a disk file.

Positioning, Reading, and Writing With Entry-Sequenced Files

Write operations to an entry-sequenced file are done by appending records to the file using the WRITE[X] procedure. Before writing, you must position the current-record and next-record pointers at the end of the file. You do this by positioning to address -1D:

```
INT(32) RECORD^ADDR;
  .
  .
RECORD^ADDR := -1D;
CALL POSITION(FILE^NUM,
              RECORD^ADDR);
CALL WRITEX(FILE^NUM,
            BUFFER,
            STRING^LENGTH);
```

To allow the appended record to be randomly accessed, you can acquire the record address (combination of block number and record number) by issuing a FILE_GETINFOLIST_ call as follows:

```
LITERAL MAX^RESULT^LENGTH = 34;
INT(32) RECORD^ADDRESS := 0D;
 .
 .
NUMBER^OF^ITEMS := 1;
GET^RECORD^ADDRESS := 12;
CALL FILE_GETINFOLIST_(FILE^NUM,
                       GET^RECORD^ADDRESS,
                       NUMBER^OF^ITEMS,
                       RECORD^ADDRESS,
                       MAX^RESULT^LENGTH);
```

Here, item 12 is passed to the procedure to return the current-record pointer. The procedure returns the current-record pointer (containing the record address) in RECORD^ADDRESS. Your program can then use this address to access the stored record using the READ[X] procedure:

```
CALL POSITION(FILE^NUM,
              RECORD^ADDRESS);

CALL READX(FILE^NUM,
           BUFFER,
           BUFFER^LENGTH,
           BYTES^READ);
```

Locking, Renaming, Caching, Closing, Purging, and Altering Entry-Sequenced Files

The operations of locking, renaming, closing, and purging entry-sequenced files, altering entry-sequenced-file attributes, and avoiding unnecessary cache flushes of entry-sequenced files are the same as for any disk file. See Using Unstructured Files earlier in this section.

Monitoring Writes to a Disk File

You can use operation 27 of the CONTROL procedure to detect write operations to a disk file. This feature is typically used with entry-sequenced files to check for writes to the end of the file.

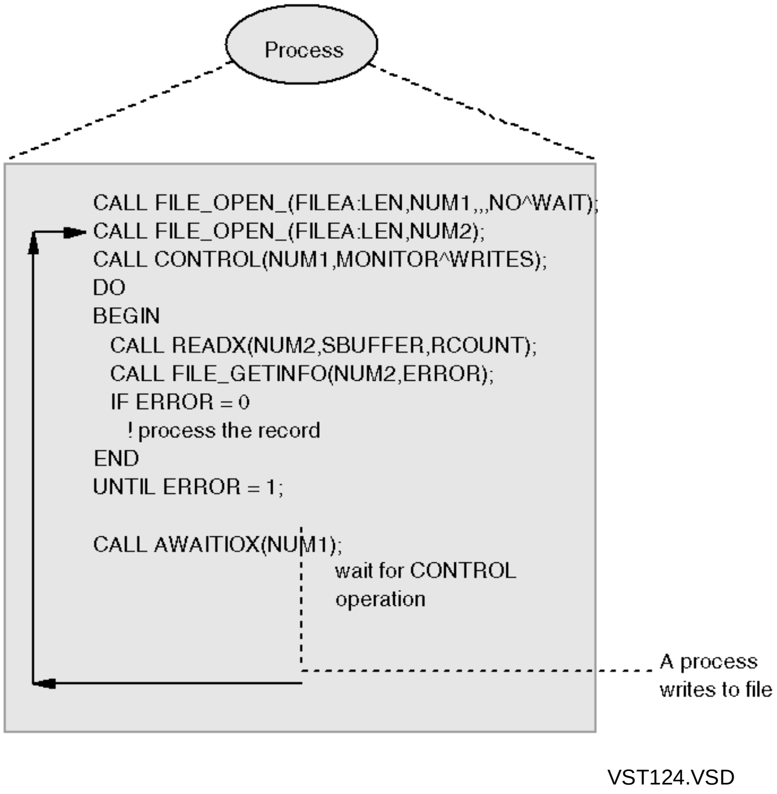
A typical example of the use of this feature is with a file that contains a log of instructions written to the file by various processes. Your process needs to read these instructions from the file and therefore needs to know when a write has taken place.

Using the CONTROL 27 Operation

To find out whether a write operation might have taken place, you issue the CONTROL 27 operation against a file number that is open for nowait I/O. When a write occurs against any open on this file, the corresponding call to the AWAITIO procedure returns. Note that adding a record to the file is not the only reason why the CONTROL operation might finish; for example, a write operation that occurs as a result of a backing out a transaction will also complete the CONTROL operation.

Figure 5-5 shows how CONTROL operation 27 is typically used.

Figure 5-5. Monitoring Write Operations on a Disk File



```
CALL FILE_OPEN_(FILEA:LEN,NUM1,,,NO^WAIT);
CALL FILE_OPEN_(FILEA:LEN,NUM2);
CALL CONTROL(NUM1,MONITOR^WRITES);
DO
BEGIN
  CALL READX(NUM2,SBUFFER,RCOUNT);
  CALL FILE_GETINFO(NUM2,ERROR);
  IF ERROR = 0
    ! process the record
END
UNTIL ERROR = 1;

CALL AWAITIOX(NUM1);
```

The following sequence explains how the example in Figure 5-5 works:

1. The first FILE_OPEN_ call opens the file for nowait I/O, with a sync-depth of 0.
2. The second FILE_OPEN_ call opens the file again, this time for waited I/O.
3. The CONTROL 27 operation is issued against the file number returned by the nowait open; the CONTROL operation returns immediately.

4. The DO-UNTIL loop issues read operations against the file number returned by the waited open. The loop exits when the READX procedure returns an end-of-file error.

   This operation serves the following purposes:

   - The first time through the loop, the calls to READX read any records that were already written to the file.
   - In subsequent loops, the READX procedure reads the record at the end of the file that was detected by the last CONTROL 27 operation that finished.
   - If the last CONTROL 27 operation finished because of some reason other than appending a record to the file, the READX procedure returns an end-of-file error on the first call. The operation that caused the CONTROL operation to finish is ignored.

5. The AWAITIO procedure returns when the CONTROL 27 operation finishes; that is, after a write operation to the file from any process has taken place.
6. The program issues another CONTROL 27 operation to wait for the next write to the file.

Using SETMODE Function 146 With CONTROL Operation 27

If several processes issue CONTROL 27 operations against the same file, the effect differs, depending on whether you use SETMODE function 146.

Several CONTROL 27 operations could be issued against the same file before there is any write to the file. In this case, all CONTROL 27 operations normally finish when one write to the file occurs. If you need only one of these processes to respond to a new record, then you can use SETMODE function 146.

When you use SETMODE function 146, a write operation completes only one of the pending CONTROL 27 operations. These operations are queued; therefore, the operation that finishes is the last one that was still pending.

Once the CONTROL 27 operation finishes for a given process, that process should read the new record and make sure no other process can read it, for example by locking it. Next time a record gets written to the file, the process that is next on the queue returns from its CONTROL 27 operation; the process skips over the locked record and reads the record that was just added.

You set the mode as follows:

```
LITERAL YES = 1,
        ONE^CONTROL27^AT^A^TIME = 146;
  .
  .
CALL SETMODE(FNUM,
             ONE^CONTROL27^AT^A^TIME,
             YES);
IF <> THEN ...
```

Entry-Sequenced File Programming Example

This example again uses the log-file program. Here, the program is shown modified to use an entry-sequenced file. The entry-sequenced file is suitable for this kind of application because:

- File entries are chronological.
- Variable-length entries are permitted. Unlike for relative files, you do not allocate 512 bytes for each record; you use only as much disk space as there are data characters entered in the record. This feature also enables entries up to a complete block in length.
- Record addresses permit random access.

Entry-sequenced files do, however, have the following drawbacks for this type of application:

- You cannot update a record with a record of arbitrary length (record updates must be exactly the same size).
- The record address is difficult to use because it is made up of the sum of the block address and the record number relative to the start of the block.

You can programmatically create the entry-sequenced file required by this program by using the FILE_CREATE_ procedure as described in Creating Entry-Sequenced Files earlier in this subsection, or you can simply use FUP commands as shown below:

```
1> FUP
-SET TYPE E
-SET BLOCK 4096
-SET REC 4072
-SHOW
    TYPE E
    EXT ( 1 PAGES, 1 PAGES )
    REC 4072
    BLOCK  4096
    MAXEXTENTS 16
-CREATE $ADMIN.OPERATOR.ESFILE
CREATED - $ADMIN.OPERATOR.ESFILE
-EXIT
2>
```

Notice that the maximum record length has been set to 4072 bytes. This is the maximum allowed in a block of 4096 bytes; the remaining 24 bytes are overhead. There is no need to restrict the record size further than this, because the only disk space used is the actual size of the data written.

The sample program shown below differs from the relative-file example as follows:

- There is no UPDATE^RECORD procedure and no corresponding option. Entry-sequenced files do not support this feature.

- The APPEND^RECORD procedure contains additional code to return the record address and display it on the terminal. The user can use this address to randomly access the record.

The code for this program follows.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST,SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !maximum file-name
                                        ! length
LITERAL BUFSIZE = 512;

STRING .SBUFFER[0:BUFSIZE];      !I/O buffer (one extra char)
STRING .S^PTR;                   !pointer to end of string
STRING .LOGNAME[0:MAXFLEN - 1];  !name of log file
INT     LOGLEN;                  !length of log name
INT     LOGNUM;                  !log file number
STRING .TERMNAME[0:MAXFLEN - 1]; !terminal file
INT     TERMLEN;                 !length of term name
INT     TERMNUM;                 !terminal file number

?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN,FILE_GETINFOLIST_)
?LIST
```

```
!------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and print
! messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Put a double-length integer into the line:

   DEFINE PUT^DOUBLE (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,N,10) #;


!  Print the line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER, @S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE (SBUFFER, 0) #;


!  Print a string:

    DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                     PUT^STR (S);
                                     PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there isn't a file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN-1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE (BUF, LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;


!-------------------------------------------------------------
! This procedure asks the user for the next function to do:
!
! "r" to read records
! "a" to append a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'a' for Append to Log, ");
   PRINT^STR("     'x' for Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                    BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r."  The start of the read is selected randomly by record
! number.  The user has the option of sequentially reading
! subsequent messages.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT          COUNT^READ;
   INT(32)      RECORD^NUM;
   STRING .EXT NEXT^ADR;
   INT          STATUS;
   INT          ERROR;

!  Prompt the user to select a record:

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Record Address:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                    BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   SBUFFER[COUNT^READ] := 0;

!  Convert ASCII to numeric:

   @NEXT^ADR := DNUMIN(SBUFFER,RECORD^NUM,10,STATUS);
   IF STATUS OR @NEXT^ADR <> $XADR(SBUFFER[COUNT^READ]) THEN
   BEGIN
      PRINT^STR("Error in the record number");
      GOTO PROMPT^AGAIN;
   END;

!  Position current-record and next-record pointers to
!  selected record:

   CALL POSITION(LOGNUM,RECORD^NUM);
   IF <> THEN  CALL FILE^ERRORS(LOGNUM);
```

```
! Loop reading and displaying records until user declines
! to read next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

      ! Read a record from the log file and display it on the
      ! terminal.  If end-of-file is reached, return control
      ! to LOGGER procedure.

      CALL READX(LOGNUM,SBUFFER,BUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(LOGNUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR ("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(LOGNUM);
      END;

      CALL WRITE^LINE(SBUFFER,COUNT^READ);

      PRINT^BLANK;

      !  Prompt the user to read the next record (user must
      !  respond "y" to accept, otherwise return to select
      !  next function):

      SBUFFER ':=' ["Do you want to read another ",
                             "record (y/n)? "]
                   -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN  CALL FILE^ERRORS(TERMNUM);

      SBUFFER[COUNT^READ] := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

```
!------------------------------------------------------------
! Procedure for appending a record.  The user selected
! function "a."  The user is prompted to enter comments.  The
! procedure puts the comments in a new record at the end of
! the file.
!------------------------------------------------------------

PROC APPEND^RECORD;
BEGIN
   INT     GET^REC^ADDR := ZSYS^VAL^FINF^CURRRECPOINTER;

   INT     COUNT^READ;
   INT     WIDTH;
   INT(32) REC^ADDR;
   INT     ERROR;

   PRINT^BLANK;

!  Prompt user for comments and read comments into the
!  buffer.

   SBUFFER ':=' "Enter today's comments:  "
                 -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                    BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

!  Place the next-record pointer at the end-of-file and
!  write the new record there:

   CALL POSITION(LOGNUM, -1D);
   IF <> THEN  CALL FILE^ERRORS(LOGNUM);

   CALL WRITEX(LOGNUM,SBUFFER,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

!  Get the record address and display it on the terminal:

   ERROR := FILE_GETINFOLIST_(LOGNUM,GET^REC^ADDR,1,
                             REC^ADDR,$LEN(REC^ADDR));
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN,ERROR);

   START^LINE;
   PUT^STR("Record address is:  ");
   PUT^DOUBLE(REC^ADDR);
   CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER);

END;
```

```
!-------------------------------------------------------------
! Procedure to exit the program.
!-------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;


!-------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "u," "a," or "x."
!-------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and then
!  return to prompt again for a valid function:

   PRINT^STR ("INVALID COMMAND: "  &
              "Type either 'r,' 'a,' or 'x'");
END;
```

```
!------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   INT     ERROR;

!  Read and discard startup messages.

   CALL INITIALIZER;

!  Open the terminal file.  For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  read from the Startup message; see Section 8, “Communicating
!  With a TACL Process,” for details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the log file with a sync depth of 1:

   LOGNAME ':=' "$ADMIN.OPERATOR.ESFILE" -> @S^PTR;
   LOGLEN := @S^PTR '-' @LOGNAME;
   ERROR := FILE_OPEN_(LOGNAME:LOGLEN,
                       LOGNUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN,ERROR);
END;
```

```
!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!-------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function "x":

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command.

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "a", "A"  -> CALL APPEND^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
   END;
END;
```

Using Key-Sequenced Files

This subsection discusses the way the programmatic interface to key-sequenced files differs from that of the other structured file types. Specifically, it discusses how to create and access a key-sequenced file. At the end of the subsection is a working example of a program that uses a key-sequenced file.

The discussion here is limited to primary-key access. Alternate keys are discussed later under the heading Using Alternate Keys.

Creating Key-Sequenced Files

You can create a key-sequenced file either interactively using the FUP CREATE command or programmatically using the FILE_CREATE[LIST]_ procedure. In either case, you need to supply information about how to build the file, such as the following:

- The file-type parameter must be set to 3 to specify a key-sequenced file.
- Key compression and compaction can also be specified on file creation. These features are used to eliminate leading or trailing parts of similar keys to save space. For details, see the *Enscribe Programmer’s Guide*.
- The block size is the number of bytes that are transferred between the disk and the disk process. The block size can be 512, 1024, 2048, or 4096 bytes (including a few bytes of overhead). Records cannot span blocks, therefore the block size must be at least large enough to contain one record plus the overhead. A block usually contains multiple records.
- The maximum record length must be set to some value within the limits imposed by the block size.
- The key length specifies the number of bytes in the primary key.
- The key offset is the number of bytes into a record where the primary key starts. The default offset is zero, the first field in the record.

The following example creates a key-sequenced file interactively using the FUP CREATE command. The new file has a block size of 4096 bytes, maximum record length of 128 bytes, and key length of 16 bytes, and the key offset is zero.

```
1> FUP
-SET TYPE K
-SET BLOCK 4096
-SET REC 128
-SET IBLOCK 2048
-SET KEYLEN 16
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 128
    BLOCK  4096
    IBLOCK 2048
    KEYLEN 16
    KEYOFF 0
    MAXEXTENTS 16
-CREATE \SYS.$MANUF.RECORDS.INVENTRY
CREATED - \SYS.$MANUF.RECORDS.INVENTRY
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

The following example creates the same key-sequenced file programmatically using the FILE_CREATE_ procedure:

```
NAME          ':=' "\SYS.$MANUF.RECORDS.INVENTRY" -> @S^PTR;
NAME^LENGTH    := @S^PTR '-' @NAME;
FILE^TYPE      := 3;
RECORD^LENGTH := 128;
BLOCK^LENGTH  := 4096;
KEY^LENGTH     := 16;
KEY^OFFSET     := 0;
CALL FILE_CREATE_(NAME:ZSYS^VAL^LEN^FILENAME,
                  NAME^LENGTH,
                  !file^code!,
                  !primary^extent^size!,
                  !secondary^extent^size!,
                  !max^extents!,
                  FILE^TYPE,
                  !options!,
                  RECORD^LENGTH,
                  BLOCK^LENGTH,
                  KEY^LENGTH,
                  KEY^OFFSET);
```

Opening Key-Sequenced Files

You open a key-sequenced file as you would any file, by using the FILE_OPEN_ procedure. See Using Unstructured Files earlier in this section for an example.

Positioning, Reading, and Writing With Key-Sequenced Files

Read and write operations on key-sequenced files are done using READ[X], READUPDATE[X], WRITE[X], and WRITEUPDATE[X] procedure calls, as for any file. However, what is unique about key-sequenced files is the way you position the current-record and next-record pointers before reading, writing, or updating the file. The KEYPOSITION procedure sets the pointers to the appropriate record using one of the positioning modes: exact, approximate, or generic.

- Exact positioning sets both pointers to the exact key value specified by the KEYPOSITION call. If there is no record with the specified key, an error is returned by the procedure that subsequently attempts to access the record. This mode is used when updating a record to be sure that the user is accessing the correct record.

  Suppose you have a file of records sorted on social security number. You would use a code fragment similar to the following to update a specific record. In this case, the code updates the record with social security number 327-67-1120.

  ```
  LITERAL EXACT = 2;
   .
   .
  KEY^VALUE ':=' "327671120" -> @S^PTR;
  KEY^LEN := @S^PTR '-' @KEY^VALUE;
  CALL KEYPOSITION(KEY^FILE^NUM,
                   KEY^VALUE,
                   !key^specifier!,
                   KEY^LEN,
                   EXACT);

  CALL READUPDATEX(KEY^FILE^NUM,
                   BUFFER,
                   $LEN(RECORD),
                   BYTES^READ);
  IF <> THEN ...
   .
   .
  CALL WRITEUPDATEX(KEY^FILE^NUM,
                    BUFFER,
                    $LEN(RECORD));
  IF <> THEN ...
   .
   .
  ```

- Approximate positioning sets both pointers to the record containing either the exact key or the next greater key. This mode is often used for starting a sequential read operation.

  For example, a user may want to examine all records starting with those whose primary key value begins with “C.” Here, you would use approximate positioning to set the pointers to the first record that begins with “C.” This example assumes that each key is made up entirely of alphabetic characters. It loops indefinitely, reading one record each time it goes through the loop. The READX procedure returns an error when the end-of-file is reached; you can use this condition to exit the loop.

```
LITERAL APPROX = 0;
 .
 .
KEY^VALUE := "CAAAAAAAAAAAAAAA";
CALL KEYPOSITION(NAME^FILE^NUM,
                 KEY^VALUE,
                 !key^specifier!,
                 !length^word!,
                 APPROX);
WHILE 1 DO
BEGIN
   CALL READX(NAME^FILE^NUM,
              BUFFER,
              $LEN(RECORD),
              BYTES^READ);
   IF <> THEN ...
END;
 .
 .
```

- Generic key positioning uses a partial key to reference a group of records that contain the partial key. If you use the key value “C” with generic positioning, then your program accesses the first record whose primary key begins with “C,” if one exists. If there is no such record, the KEYPOSITION call returns without error but the I/O operation that attempts to access the record does return an error.

  To use generic key positioning, you must also supply the length of the part of the key that will be used to start the generic access. In the example given below, the single letter “C” is used, therefore the key length is set to 1.

  In the following example, a READX call returns an end-of-file indication as soon as the key value no longer matches the generic key given in the KEYPOSITION call:

```
LITERAL GENERIC = 1;
 .
 .
KEY^VALUE := "C";
KEY^LENGTH := 1;
CALL KEYPOSITION(NAME^FILE^NUM,KEY^VALUE,
                 !key^specifier!,
                 KEY^LENGTH,GENERIC);
WHILE 1 DO
BEGIN
   CALL READX(NAME^FILE^NUM,BUFFER,
              $LEN(RECORD),BYTES^READ);
   IF <> THEN ...
END;
 .
 .
```

So far, sequential reading of a key-sequenced file has been assumed to mean reading records in ascending key sequence. By setting bit 1 of the *`positioning-mode`* parameter, however, you can read sequentially in descending key sequence.

Positioning is unnecessary when writing a new record to a key-sequenced file. The Enscribe software responds to the WRITEX call by inserting the new record into the file in position according to its key value.

Locking, Renaming, Caching, Closing, Purging, and Altering Key-Sequenced Files

The operations of locking, renaming, closing, and purging key-sequenced files, altering key-sequenced-file attributes, and avoiding unnecessary cache flushes of key-sequenced files are the same as for any disk file. See Using Unstructured Files earlier in this section.

Key-Sequenced File Programming Example

A different application will be used to illustrate the use of key-sequenced files. This example provides access to a key-sequenced file that contains an inventory. Information about each item is stored in a record accessible by part number. The record structure is as follows:

part number	description	desc-len	supplier	supp-len	quantity	price
6 bytes	60 bytes	2 bytes	60 bytes	2 bytes	2 bytes	2 bytes

CDT 027.CDD

You can create the key-sequenced file required by this program using the FILE_CREATE[LIST]_ procedure as described under Creating Key-Sequenced Files earlier in this section. Or you can simply use FUP commands as shown below:

```
1> FUP
-SET TYPE K
-SET BLOCK 2048
-SET REC 134
-SET IBLOCK 2048
-SET KEYLEN 6
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 134
    BLOCK  2048
    IBLOCK 2048
    KEYLEN 6
    KEYOFF 0
    MAXEXTENTS 16
-CREATE $APPL.SUBAPPL.PARTFILE
CREATED - $APPL.SUBAPPL.PARTFILE
```

```
-EXIT
2>
```

The program is similar to the relative-file example given earlier in this section in that it enables the user to read records, add records, and update records. Because access to the file is by key value, however, the mechanism is different.

The following procedures provide the major functions of reading, updating, and inserting records:

- The READ^RECORD procedure allows the user to read one record followed optionally by subsequent sequential reads as it did in the relative-file program. But here, the key to the random record is the part number—a field of data in the record itself, not the physical record number. Also, because of the way Enscribe manages key-sequenced files, sequential reading returns records in key sequence (by part number), not physical sequence.

  This procedure uses an approximate key position to enable the user to start reading from a particular key value without concern as to whether the key actually exists. This feature enables the user to start browsing the file from any key value.

- The UPDATE^RECORD procedure displays the record for update before prompting the user for the updated information. First it prompts the user for the key to the record to be updated (the part number). Then it uses the READUPDATEX procedure to get the current information from the record. After displaying the current contents of the record on the user's terminal and receiving the new contents from the user, this procedure reads the record from the disk file again, this time using the READUPDATELOCKX procedure; in addition to reading the record to check whether the record has been modified by some other user since the previous READUPDATEX call, this procedure also locks the record to ensure exclusive access while updating. Finally, UPDATE^RECORD issues a call to WRITEUPDATEUNLOCKX to write the new record contents to disk and unlock the record.

  Locking and unlocking the record protects the record against other processes while your process is updating the record. Multiple copies of this program can therefore exist without corrupting each other’s view of data.

- The INSERT^RECORD procedure replaces the APPEND^RECORD procedure of the log-file program. INSERT^RECORD allows the user to insert new records into the file. Here, the procedure prompts for the contents of the new record (including the part number) and then writes the record in the appropriate position in the file. The insertion is rejected if a record with the same key already exists.

The following procedures support the above major procedures:

- The DISPLAY^RECORD procedure displays the contents of a part record.

- The ENTER^RECORD procedure prompts for information from the user to create a record. When creating a new record, this procedure prompts for every field in the new record. When updating an existing record, this procedure prompts for all but the part number, which is already known. The parameter to the procedure specifies whether an update or a new record is required.

The code for this program appears on the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST,SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;  !maximum file-name
                                          ! length
LITERAL OLD     = 0;                !updating in ENTER^REC
LITERAL NEW     = 1;                !new record in ENTER^REC
LITERAL BUFSIZE = 132;              !size of terminal buffer
LITERAL PARTSIZE= 6;                !size of part number
LITERAL DESCSIZE= 60;               !size of part description
LITERAL SUPPSIZE= 60;                  !size of supplier name

STRING .SBUFFER[0:BUFSIZE];         !I/O buffer (one extra
                                    ! char)
STRING .S^PTR;                      !pointer to end of string
INT     PARTFILE^NUM;               !part file number
INT     TERMNUM;                    !terminal file number

STRUCT .PART^RECORD;
BEGIN
   STRING PART^NUMBER[0:PARTSIZE-1];
   STRING DESCRIPTION[0:DESCSIZE-1];
   INT    DESC^LEN;
   STRING SUPPLIER[0:SUPPSIZE-1];
   INT    SUP^LEN;
   INT    ON^HAND;
   INT    UNIT^PRICE;
END;

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,NUMIN,
?  KEYPOSITION,PROCESS_STOP_,READX,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN,READUPDATELOCKX,
?  WRITEUPDATEUNLOCKX,FILE_GETINFOLIST_,UNLOCKREC)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

   DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, when there is no file number for it.
! FILE^ERRORS is used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING .FNAME;
INT     LEN;
INT     ERROR;
BEGIN

!  Compose and print the message

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS (FNUM);
INT         FNUM;
BEGIN
   INT      ERROR;
   STRING  .FNAME[0:MAXFLEN - 1];
   INT      FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!-------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!-------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING  .BUF;
INT      LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
END;


!-------------------------------------------------------------
! This procedure asks the user for the next function to do:
!
! "r" to read records
! "u" to update a record
! "i" to insert a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' to Read Record, ");
   PRINT^STR("     'u' to Update a Record,  ");
   PRINT^STR("     'i' to Insert a Record, ");
   PRINT^STR("     'x' to Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER, @S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to display a part record on the terminal
!------------------------------------------------------------

PROC DISPLAY^RECORD;
BEGIN

   PRINT^BLANK;

!  Display part number:

   PRINT^STR("Part Number Is:    " & PART^RECORD.PART^NUMBER
                                    FOR PARTSIZE);

!  Display part description:

   PRINT^STR("Part Description:  " & PART^RECORD.DESCRIPTION
                                    FOR PART^RECORD.DESC^LEN);

!  Display part supplier name:

   PRINT^STR("Supplier:          " & PART^RECORD.SUPPLIER
                                    FOR PART^RECORD.SUP^LEN);

!  Display quantity on hand:

   START^LINE;
   PUT^STR("Quantity on hand:  ");
   PUT^INT(PART^RECORD.ON^HAND);
   PRINT^LINE;

!  Display unit price:

   START^LINE;
   PUT^STR("Unit Price:       $");
   PUT^INT(PART^RECORD.UNIT^PRICE);
   PRINT^LINE;
END;
```

```
!------------------------------------------------------------
! Procedure to prompt user for input to build a new record or
! update an existing record.  When updating, an empty
! response (COUNT^READ=0) means to leave the existing value
! unchanged.
!------------------------------------------------------------

PROC ENTER^RECORD(TYPE);
INT  TYPE;

BEGIN
   INT     COUNT^READ;
   INT     STATUS;
   STRING .NEXT^ADDR;

   DEFINE BLANK^FILL(F) =
            F ':=' " " & F FOR $LEN(F)*$OCCURS(F)-1 BYTES #;

   PRINT^BLANK;

!  If inserting a new record, prompt for a part number.
!  If updating an existing record, record number is already
!  known:

   IF TYPE = NEW THEN
   BEGIN
      SBUFFER ':=' "Enter Part Number:  " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      BLANK^FILL(PART^RECORD.PART^NUMBER);
      PART^RECORD.PART^NUMBER ':='
                     SBUFFER FOR $MIN(COUNT^READ,PARTSIZE);
   END;

!  Prompt for a part description:

   SBUFFER ':=' "Enter Part Description:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,DESCSIZE);
      BLANK^FILL(PART^RECORD.DESCRIPTION);
      PART^RECORD.DESCRIPTION ':=' SBUFFER FOR COUNT^READ;
      PART^RECORD.DESC^LEN := COUNT^READ;
   END;
```

```
!  Prompt for the name of the supplier:

   SBUFFER ':=' "Enter Supplier Name:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,SUPPSIZE);
      BLANK^FILL(PART^RECORD.SUPPLIER);
      PART^RECORD.SUPPLIER ':=' SBUFFER FOR COUNT^READ;
      PART^RECORD.SUP^LEN := COUNT^READ;
   END;

!  Prompt for the quantity on hand and unit price:

PROMPT^AGAIN:
   SBUFFER ':=' "Enter Quantity On Hand:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR :=
         NUMIN(SBUFFER,PART^RECORD.ON^HAND,10,STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN;
      END;
   END;

PROMPT^AGAIN1:
   SBUFFER ':=' "Enter Unit Price:  $" -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR :=
         NUMIN(SBUFFER,PART^RECORD.UNIT^PRICE,10,STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN1;
      END;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r." The start of the read is selected by approximate key
! positioning.  The user has the option of sequentially
! reading subsequent records.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT          COUNT^READ;
   INT          ERROR;

!  Prompt the user for the part number:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);


!  Position approximately to the selected record:

   CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,
                    !key^specifier!,
                    COUNT^READ,0);
   IF <> THEN  CALL FILE^ERRORS(PARTFILE^NUM);

!  Loop reading and displaying records until user declines
!  to read the next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Read a record from the part file.
   !  If end-of-file is reached,
   !  return control to the main procedure.

      CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(PARTFILE^NUM);
      END;

   !  Display the record on the terminal:

      CALL DISPLAY^RECORD;
```

```
         PRINT^BLANK;

      !  Prompt the user to read the next record (user
      !  must respond "y" to accept, otherwise return
      !  to select next function):

         SBUFFER ':=' ["Do you want to read another ",
                                  "record (y/n)? "]
                      -> @S^PTR;
         CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                          BUFSIZE,COUNT^READ);
         IF <> THEN CALL FILE^ERRORS(TERMNUM);

         SBUFFER[COUNT^READ] := 0;
      END
      UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
   END;
```

```
!------------------------------------------------------------
! Procedure for updating a record.  The user selected
! function "u."  The user is prompted to enter the part
! number of the record to be updated, then the old contents
! are displayed on the user's terminal before the user
! is prompted to enter the updated record.
!------------------------------------------------------------

PROC UPDATE^RECORD;
BEGIN

   INT     COUNT^READ;
   INT     ERROR;
   STRUCT .SAVE^REC(PART^RECORD);
   STRUCT .CHECK^REC(PART^RECORD);

   PRINT^BLANK;

!  Prompt the user for the part number of the record to be
!  updated:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);


!  Position exactly to the selected record.

!   SBUFFER[COUNT^READ] ':=' [PARTSIZE*[" "]];
   CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,
                    !key^specifier!,
                    COUNT^READ,2);
   IF <> THEN  CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record.  If no such record exists,
!  the procedure informs the user and returns control to
!  the main procedure:

   CALL READUPDATEX(PARTFILE^NUM,PART^RECORD,
                    $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
      IF ERROR = 11 THEN
      BEGIN
         PRINT^BLANK;
         START^LINE;
         PUT^STR("No such record");
         PRINT^LINE;
         RETURN;
      END
      ELSE CALL FILE^ERRORS(PARTFILE^NUM);
   END;
```

```
!  Save the record for later comparison

   SAVE^REC ':=' PART^RECORD FOR $LEN(PART^RECORD) BYTES;

!  Display the record on the terminal:

   CALL DISPLAY^RECORD;

!  Prompt the user for the updated record:

   CALL ENTER^RECORD(OLD);

!  Now that we have the user's changes, reread the record
!  and check to see whether someone else changed it while
!  the user was responding.

   CALL READUPDATELOCKX(PARTFILE^NUM,CHECK^REC,
                        $LEN(PART^RECORD));
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

   IF CHECK^REC <> SAVE^REC FOR $LEN(PART^RECORD) BYTES THEN
   BEGIN
      CALL UNLOCKREC(PARTFILE^NUM);
      PRINT^BLANK;
      PRINT^STR("The record was changed by someone else " &
                "while you were working on it.");
      PRINT^STR("Your change was not made.");
      RETURN;
   END;

!  Write the new record to the file:

   CALL WRITEUPDATEUNLOCKX(PARTFILE^NUM,PART^RECORD,
                           $LEN(PART^RECORD));
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);
END;
```

```
!------------------------------------------------------------
! Procedure for inserting a record.  The user selected
! function "i." The user is prompted to enter the new record.
! The procedure inserts the new record in the appropriate
! place in the file.
!------------------------------------------------------------

PROC INSERT^RECORD;
BEGIN
   INT     ERROR;

   PRINT^BLANK;

!  Prompt the user for the new record:

   CALL ENTER^RECORD(NEW);

!  Write the new record to the file:

   CALL WRITEX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
      IF ERROR = 10 THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR
            ("There is already a record with that " &
             "part number.");
         PRINT^STR("Your new one was not entered.");
      END ELSE BEGIN
         CALL FILE^ERRORS(PARTFILE^NUM);
      END;
   END;
END;


!------------------------------------------------------------
! Procedure to exit the program.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "u", "a," or "x."
!------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that his selection was invalid
!  then return to prompt again for a valid function:

   PRINT^STR("INVALID COMMAND: "  &
             "Type either 'r,' 'u,' 'i,' or 'x'");
END;


!------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .PARTFILE^NAME[0:MAXFLEN - 1]; !name of part file
   INT     PARTFILE^LEN;                 !length of part-file
                                         ! name
   STRING .TERMNAME[0:MAXFLEN - 1];      !terminal file
   INT     TERMLEN;                      !length of terminal-
                                         ! file name
   INT     ERROR;

!  Read and discard startup messages.

   CALL INITIALIZER;

!  Open the terminal file. For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  read from the Startup message; see Section 8, Communicating 
With a TACL Process for
!  details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
```

```
!  Open the part file with a sync depth of 1:

   PARTFILE^NAME ':=' "$APPL.SUBAPPL.PARTFILE" -> @S^PTR;
   PARTFILE^LEN := @S^PTR '-' @PARTFILE^NAME;
   ERROR := FILE_OPEN_(PARTFILE^NAME:PARTFILE^LEN,
                       PARTFILE^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(PARTFILE^NAME:PARTFILE^LEN,
                            ERROR);
END;


!------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize, then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!------------------------------------------------------------

PROC PARTS MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function x:

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command.

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "u", "U"  -> CALL UPDATE^RECORD;

         "i", "I"  -> CALL INSERT^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
```

```
       END;
    END;
```

```
!-------------------------------------------------------------
! Procedure for inserting a record.  The user selected
! function "i."  The user is prompted to enter comments.  The
! procedure puts the comments in a new record at the end of
! the file.
!-------------------------------------------------------------

PROC INSERT^RECORD;
BEGIN
   INT    COUNT^READ;
   INT    ERROR;

   PRINT^BLANK;

!  Prompt user for comments and read comments into the
!  buffer:

   CALL GET^DATE;
   RECORD.DATE ':=' SBUFFER FOR DATESIZE;

   SBUFFER ':=' "Enter comments:  "
                 -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   COMMENTSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   RECORD.DATA ':=' SBUFFER FOR COUNT^READ;
   RECORD.DATA^LEN := COUNT^READ;

!  Position to the end of file and write the new record:

   CALL POSITION(LOGNUM,-1D);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

   CALL WRITEX(LOGNUM,RECORD,$LEN(RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(LOGNUM,ERROR);
      IF ERROR = 10 THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR
            ("There is already a record for that date.");
         PRINT^STR("This comment was not entered.");
      END ELSE BEGIN
         CALL FILE^ERRORS(LOGNUM);
      END;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to exit the program.
!-------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;


!-------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "u," "i," or "x."
!-------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and then
!  return to prompt again for a valid function:

   PRINT^STR("INVALID COMMAND: "  &
             "Type either 'r,' 'u,' 'i,' or 'x'");
END;
```

```
!-------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .LOGNAME[0:MAXFLEN - 1];  !name of log file
   INT     LOGLEN;                  !file name length
   STRING .TERMNAME[0:MAXFLEN - 1]; !terminal file
   INT     TERMLEN;                 !name length
   INT     ERROR;

!  Read and discard startup sequence of messages:

   CALL INITIALIZER;

!  Open the terminal file. For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  read from the Startup message; see Section 8,
!  "Communicating With the TACL Process," for details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the log file with a sync depth of 1:

   LOGNAME ':=' "$ADMIN.OPERTOR.ALTLOG" -> @S^PTR;
   LOGLEN := @S^PTR '-' @LOGNAME;
   ERROR := FILE_OPEN_(LOGNAME:LOGLEN,
                       LOGNUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(LOGNAME:LOGLEN,ERROR);
END;
```

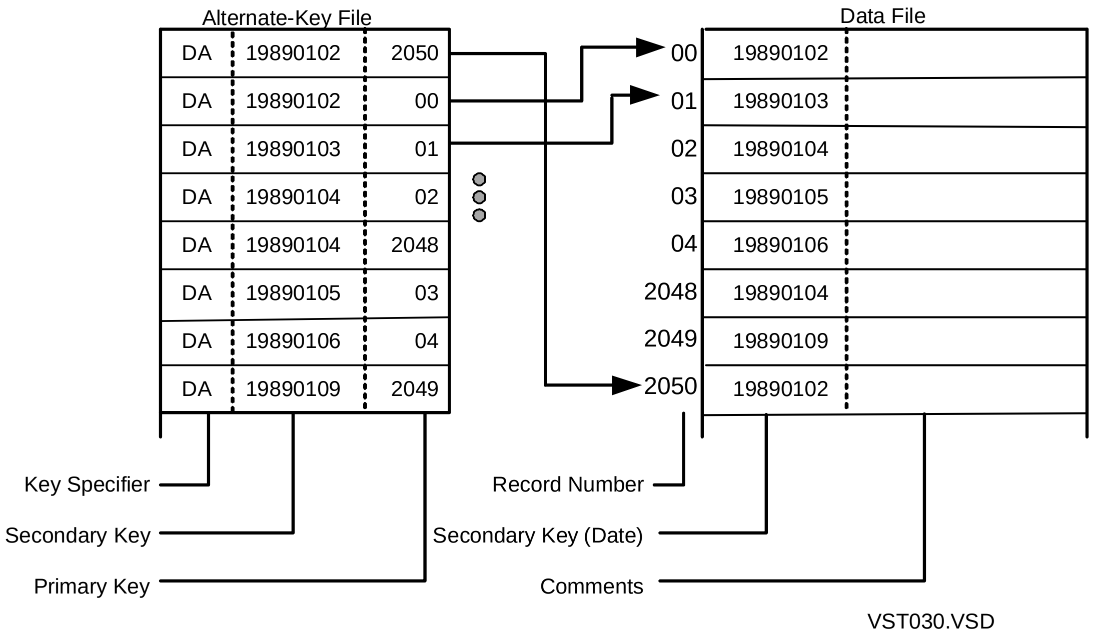
```
!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!-------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until the user selects function "x":

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command:

      CMD := GET^COMMAND;

   !  Call the function selected by the user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "u", "U"  -> CALL UPDATE^RECORD;

         "i", "I"  -> CALL INSERT^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
   END;
END;
```

Using Alternate Keys With an Entry-Sequenced File

The application of alternate keys to an entry-sequenced file is similar to applying alternate keys to relative files. Instead of a record number, the alternate-key file cross-references the alternate-key value to a record address.

Applying alternate keys to the example given in the subsection Using Entry-Sequenced Files produces the structure shown in Figure 5-6:

Figure 5-6. Example of Alternate-Key File for Use With an Entry-Sequenced File


Alternate-Key File
DA 19890102 2050
DA 19890102 00
DA 19890103 01
DA 19890104 02
DA 19890104 2048
DA 19890105 03
DA 19890106 04
DA 19890109 2049
Data File
00 19890102
01 19890103
02 19890104
03 19890105
04 19890106
2048 19890104
2049 19890109
2050 19890102
Key Specifier
Secondary Key
Primary Key
Record Number
Secondary Key (Date)
Comments
VST030.VSD

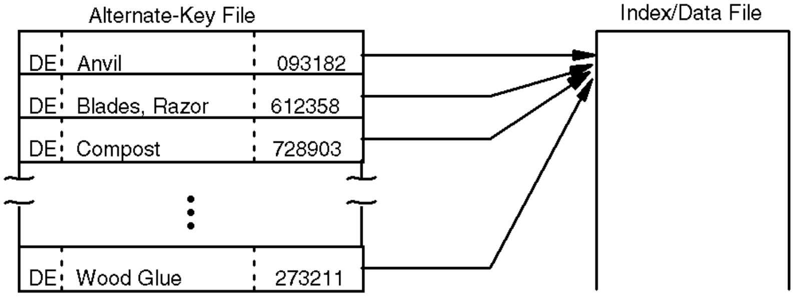
You can enhance the sample program shown in the subsection Using Entry-Sequenced Files to use the date as an alternate key by making exactly the same changes as were made to the relative-file example.

Using Alternate Keys With a Key-Sequenced File

When using alternate keys with a key-sequenced file, the alternate key cross-references the primary key.

For an example of how to use alternate keys with a key-sequenced file, the example given in the subsection Using Key-Sequenced Files is modified to be able to read records using the part description as a key. The alternate-key file therefore lists records in part-description order, each referencing a part number. Figure 5-7 shows the sample file structure.

Figure 5-7. Example of Alternate-Key File for Use With a Key-Sequenced File



VST031.VSD

To create the data and alternate-key files, you can use the FILE_CREATE_ procedure as described under Creating Alternate-Key Files earlier in this subsection, or you can use the FUP utility as follows:

```
1> FUP
-SET TYPE K
-SET REC 134
-SET BLOCK 4096
-SET IBLOCK 4096
-SET KEYLEN 6
-SET ALTKEY("DE",KEYOFF 6,KEYLEN 60)
-SET ALTFILE(0, ALT2)
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 130
    BLOCK 4096
    IBLOCK 4096
    KEYLEN 6
    KEYOFF 0
    ALTKEY( "DE", FILE 0, KEYOFF 6, KEYLEN 60)
    ALTFILE(0, $ADMIN.OPERATOR.ALT2)
    ALTCREATE
    MAXEXTENTS 16
-CREATE KEY2FILE
CREATED $ADMIN.OPERATOR.KEY2FILE
CREATED $ADMIN.OPERATOR.ALT2
-EXIT
2>
```

Few changes need to be made to the old program, because the same record structure as before is used. The only change is in the READ^RECORD procedure, which now prompts the user whether the access is to be by part number or by part description. If the user chooses to access by part number, then the procedure prompts for the part number as before. If the user chooses to access the file by part description, then the procedure uses “DE” (short for “description”) as the key specifier. Because the part description is variable in length, the key length is specified in the call to KEYPOSITION.

You can add further options to the following code to access records by supplier name, inventory level, or price.

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r."  The start of the read is selected by approximate key
! positioning.  The user has the option of sequentially
! reading subsequent records.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT         COUNT^READ;
   INT         ERROR;
   INT         KEY^SPEC;
   INT         POSITIONING^MODE;
   INT         COMPARE^LEN;

   LITERAL     APPROX  = 0;
   LITERAL     GENERIC = 1;

!  Prompt the user for the key to access the record by:

   PRINT^BLANK;
   PRINT^STR("Type 'p' to access by part number");
   PRINT^STR("Type 'd' to access by part description");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER, @S^PTR '-' @SBUFFER,
                  BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   CASE SBUFFER[0] OF
   BEGIN
      "p", "P" ->

      !  Prompt the user for the part number:

         PRINT^BLANK;
         SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
         CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                        BUFSIZE,COUNT^READ);
         IF <> THEN  CALL FILE^ERRORS(TERMNUM);

      !  Set the key specifier to zero for the primary key:

         KEY^SPEC := 0;
```

```
        !  Set the compare length to the length of the
        !  primary key and pad the value with blanks, in case
        !  the full length was not entered:

           COMPARE^LEN := PARTSIZE;
           SBUFFER[COUNT^READ] ':=' [PARTSIZE*[" "]];

        !  Set the positioning mode to approximate:

           POSITIONING^MODE := APPROX;

        "d", "D" ->

        !  Prompt for part description:

           PRINT^BLANK;
           SBUFFER ':=' "Enter Part Description:  " -> @S^PTR;
           CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                           BUFSIZE,COUNT^READ);
           IF <> THEN CALL FILE^ERRORS(TERMNUM);

        !  Set key specifier to "DE":

           KEY^SPEC := "DE";

        !  Set positioning mode to generic:

           POSITIONING^MODE := GENERIC;

        !  Set the compare length equal to the number of
        !  bytes entered:

           COMPARE^LEN := COUNT^READ;

        OTHERWISE ->
           PRINT^STR("Invalid key");
           RETURN;
     END;


  !  Position to the selected record:

     CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,KEY^SPEC,
                      COMPARE^LEN,POSITIONING^MODE);
     IF <> THEN  CALL FILE^ERRORS(PARTFILE^NUM);
```

```
!  Loop reading and displaying records until user declines
!  to read the next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Read a record from the part file.
   !  If the end of file is reached,
   !  return control to the main procedure.

      CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(PARTFILE^NUM);
      END;

   !  Display the record on the terminal:

      CALL DISPLAY^RECORD;

      PRINT^BLANK;

   !  Prompt the user to read the next record (user
   !  must respond "y" to accept, otherwise return
   !  to select next function):

      SBUFFER ':=' ["Do you want to read another ",
                                "record (y/n)? "]
                     -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN  CALL FILE^ERRORS(TERMNUM);

      SBUFFER[COUNT^READ] := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

Using Partitioned Files

When you create a file, you can choose to have the file reside on multiple volumes. A file can span up to 16 volumes in this way. Moreover, the disk volumes can be connected to the same or different controllers, on the same or different processing modules, or can even span multiple systems. Once the file is created, the locations of file partitions are transparent to the application. The user does not need to be concerned about which partition to access; the user sees the file as contiguous.

The obvious reason for partitioning a file is to acquire more disk space for a file that does not fit on one disk volume. Partitioning files, however, can also improve application performance by taking advantage of parallelism:

- If the file resides on several volumes connected to the same controlling device, then disk-head movements (or “seek” operations) can overlap on the different disk drives.
- If each partition resides on a volume that is connected to a different controller, then data transfers can occur concurrently.
- If each partition resides on a volume connected to a different processing module, then concurrent processing is possible.

For relative and entry-sequenced files, the application uses the primary partition until it is full, then starts to fill up the first extra partition, and so on. For key-sequenced files, you assign partial key values to each partition when the file is created; for example, the first partition might contain keys A through J, the second partition keys K through Q, and the third partition keys R through Z.

Creating Partitioned Files

To create a partitioned file, you need only create the primary partition; the extra partitions are created automatically, assuming you give the system the correct information. You can create a partitioned file either interactively using the FUP CREATE command or programmatically using the FILE_CREATELIST_ procedure.

Note. Use care when naming your secondary partitions. For secondary partitions that reside on a remote system with respect to the primary partition, you can use only names that have 7 characters or fewer, because one byte of the name is used to contain the node number. Secondary partitions that reside on the same node as the primary partition can have up to 8 characters.

The simplest way to create a partitioned file is by using the FUP CREATE command as shown below. This example creates a partitioned file that could be used by the inventory application described in the subsection Using Key-Sequenced Files, earlier in this section.

```
1>
FUP
-SET TYPE K
-SET REC 134
-SET EXT (64,8)
-SET BLOCK 4096
-SET IBLOCK 4096
-SET KEYLEN 8
-SET PART (1,$PART1,64,8,"25")
-SET PART (2,$PART2,64,8,"50")
-SET PART (3,$PART3,64,8,"75")
-CREATE KEYFILE
 .
 .
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

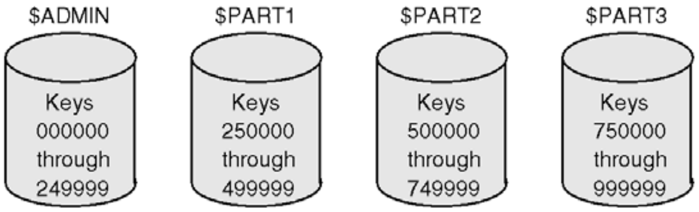
The next example creates the same file programmatically using the FILE_CREATELIST_ procedure. You supply this procedure with an *`item-list`* and a list of corresponding values. The *`item-list`* parameter is an array of numbers that identify the values given in the *`values`* parameter. In the following example, item number 41 identifies the first word of the *`values`* array as the file type, item number 43 identifies the second word as the record length, and so on.

```
?NOLIST
?INSPECT,SYMBOLS
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL

!  Global variables:

STRING .S^PTR;
 .
 .

?SOURCE $SYSTEM.SYSTEM.EXTDECS0
?LIST
 .
 .
```

```
PROC CREATE^PARTS MAIN;
BEGIN
   STRING .KEYFILE[0:ZSYS^VAL^LEN^FILENAME];  !primary-key
                                              ! file name
   INT     LENGTH;                  !length of primary file name
   INT    .ITEM^LIST[0:63];         !list of items to pass to
                                    ! FILE_CREATELIST_
   INT    .VALUES[0:512];           !values of those items
   INT     NUMBER^ITEMS;            !number of items
   INT     VALUES^LEN;              !total length of items
   INT     ERROR;                   !system procedure call error

   KEYFILE ':=' "$ADMIN.OPERATOR.KEYFILE" -> @S^PTR;

   LENGTH := @S^PTR '-' @KEYFILE;

   ITEM^LIST ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                   ZSYS^VAL^FCREAT^LOGICALRECLEN,
                   ZSYS^VAL^FCREAT^BLOCKLEN,
                   ZSYS^VAL^FCREAT^KEYOFFSET,
                   ZSYS^VAL^FCREAT^KEYLEN,
                   ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                   ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                   ZSYS^VAL^FCREAT^NUMPRTNS,
                   ZSYS^VAL^FCREAT^PRTNDESC,
                   ZSYS^VAL^FCREAT^PRTNVOLLEN,
                   ZSYS^VAL^FCREAT^PRTNVOLNAMES,
                   ZSYS^VAL^FCREAT^PRTNPARTKEYLEN,
                   ZSYS^VAL^FCREAT^PRTNPARTKEYVAL];

   NUMBER^ITEMS := 13;

   VALUES ':='    [3,     !primary-key file type
                   134,   !primary-key file record length
                   4096,  !primary-key file block length
                   0,     !primary-key file key offset
                   6,     !primary-key file key length
                   64,    !number of alternate-key specifiers
                   8,
                   64,8,64,8,64,8,
                   6,6,6,
                   "$PART1$PART2$PART3",
                   2,
                   "255075"] -> @S^PTR;

   VALUES^LEN := (@S^PTR '-' @VALUES) '<<' 1;  !length in
                                               ! bytes
                                               ! of VALUES
                                               ! parameter
!  Create the file:

   ERROR := FILE_CREATELIST_(KEYFILE:ZSYS^VAL^LEN^FILENAME,
                             LENGTH,
                             ITEM^LIST,
                             NUMBER^ITEMS,
                             VALUES,
                             VALUES^LEN);
END;
```

Either of the above examples creates a file with four partitions on volumes $ADMIN, $PART1, $PART2, and $PART3. The records are segregated by key value as shown in Figure 5-8:

Figure 5-8. Sample Partitioned File


$ADMIN
Keys 000000 through 249999
$PART1
Keys 250000 through 499999
$PART2
Keys 500000 through 749999
$PART3
Keys 750000 through 999999

VST032.VSD

Accessing Partitioned Files

You access a partitioned file in exactly the same way you would a nonpartitioned file. Open the file by simply opening the file name of the primary partition. The FILE_OPEN_ procedure returns just one file number, which you use to access the file as you would any other disk file.

Using Alternate Keys

This subsection examines how you can access a record in a structured file using a key value other than the primary key. To do this, you need an alternate-key file to provide the link between the alternate and primary keys.

Creating Alternate-Key Files

You can create primary-key and alternate-key files either interactively using the FUP CREATE command or programmatically using the FILE_CREATELIST_ procedure. In either case, you need to supply information about how to build the files, including the attributes of the primary file as well as the attributes of the alternate-key file.

The simplest way to create alternate-key files is by using the FUP CREATE command as shown below. This example creates a primary-key file like the one created in the previous example and also creates two alternate-key files to access the data records by part description ("DE" specifier) and supplier name ("SU" specifier).

```
1> FUP
-SET TYPE R
-SET BLOCK 4096
-SET REC 130
-SET KEYOFF 0
-SET KEYLEN 8
-SET ALTKEY ("DE",FILE 0,KEYOFF 8,KEYLEN 60)
-SET ALTFILE (0,$ADMIN.OPERATOR.ALT2)
-SET ALTKEY ("SU",FILE 1,KEYOFF 66,KEYLEN 60)
-SET ALTFILE (1,$ADMIN.OPERATOR.ALT3)
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 130
    BLOCK  4096
    ALTKEY ( "DE", FILE 0, KEYOFF 6, KEYLEN 60 )
    ALTKEY ( "SU", FILE 1, KEYOFF 66, KEYLEN 60 )
    ALTFILE ( 0, $ADMIN.OPERATOR.ALT2 )
    ALTFILE ( 1, $ADMIN.OPERATOR.ALT3 )
    ALTCREATE
    MAXEXTENTS 16
-CREATE KEY2FILE
CREATED - $ADMIN.OPERATOR.KEY2FILE
CREATED - $ADMIN.OPERATOR.ALT2
CREATED - $ADMIN.OPERATOR.ALT3
 .
 .
-EXIT
2>
```

Refer to the *File Utility Program (FUP) Reference Manual* for more details on how to create files using the FUP CREATE command.

The next example creates the same files programmatically using the FILE_CREATELIST_ procedure. You supply this procedure with an *`item-list`* and a list of corresponding values. The *`item-list`* parameter is an array of numbers that identify the values given in the *`values`* parameter. In the example below, item number 41 identifies the first word of the *`values`* array as the file type, item number 43 identifies the second word as the record length, and so on.

```
?NOLIST
?INSPECT,SYMBOLS
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL

!  Global variables:

STRING .S^PTR;
 .
 .

?SOURCE $SYSTEM.SYSTEM.EXTDECS0
?LIST
 .
 .

PROC CREATE^ALTS MAIN;
BEGIN
   STRING .KEYSFILE[0:ZSYS^VAL^LEN^FILENAME];  !primary-key
                                               ! file name
   INT     LENGTH;               !length of primary file name
   INT    .ITEM^LIST[0:63];      !list of items to pass to
                                 ! FILE_CREATELIST_
   INT    .VALUES[0:512];        !values of those items
   INT     NUMBER^ITEMS;         !number of items
   INT     VALUES^LEN;           !total length of items
   INT     REC^LEN;              !alternate-key file record
                                 ! length
   INT     BLOCK^LEN;            !alternate-key file block
                                 ! length
   INT     KEY^LEN;              !alternate-key file key
                                 ! length
   INT     KEY^OFFSET;           !alternate-key file key
                                 ! offset
   INT     ERROR;                !system procedure call error

   KEYSFILE ':=' "$ADMIN.OPERATOR.KEY2FILE" -> @S^PTR;

   LENGTH := @S^PTR '-' @KEYSFILE;
```

```
ITEM^LIST ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                ZSYS^VAL^FCREAT^LOGICALRECLEN,
                ZSYS^VAL^FCREAT^BLOCKLEN,
                ZSYS^VAL^FCREAT^KEYOFFSET,
                ZSYS^VAL^FCREAT^KEYLEN,
                ZSYS^VAL^FCREAT^NUMALTKEYS,
                ZSYS^VAL^FCREAT^ALTKEYDESC,
                ZSYS^VAL^FCREAT^NUMALTKEYFILES,
                ZSYS^VAL^FCREAT^ALTFILELEN,
                ZSYS^VAL^FCREAT^ALTFILENAMES];

NUMBER^ITEMS := 10;

VALUES ':='    [3,     !primary-key file type
                130,   !primary-key file record length
                4096,  !primary-key file block length
                0,     !primary-key file key offset
                6,     !primary-key file key length
                2,     !number of alternate-key specifiers

             !  Alternate key descriptor for description
             !  field:

                "DE",  !key specifier for description
                       ! field
                60,    !length of alternate key
                6,     !alternate-key file key offset
                0,     !alternate-key file number
                0,     !not used
                0,     !not used

             !  Alternate key descriptor for supplier
             !  field:

                "SU",  !key specifier for description
                       ! field
                60,    !length of alternate key
                66,    !alternate-key file key offset
                0,     !alternate-key file number
                0,     !not used
                0,     !not used

             !  Other values:

                2,     !number of alternate-key files
                20,    !length of first alternate-key
                       ! file name
                20,    !length of second alternate-key
                       ! file name
                !  Concatenated alternate-key file names:

                   "$ADMIN.OPERATOR.ALT2
                   $ADMIN.OPERATOR.ALT3"]
                                                  -> @S^PTR;

   VALUES^LEN := (@S^PTR '-' @VALUES) '<<' 1;  !length in
                                               ! bytes
                                               ! of VALUES
                                               ! parameter

!  Create the primary file:

   ERROR := FILE_CREATELIST_(KEYSFILE:ZSYS^VAL^LEN^FILENAME,
                             LENGTH,
                             ITEM^LIST,
                             NUMBER^ITEMS,
                             VALUES,
                             VALUES^LEN);

!  Create the alternate-key file ALT2:

   KEYSFILE ':=' "$ADMIN.OPERATOR.ALT2" -> @S^PTR;
   LENGTH := @S^PTR '-' @KEYSFILE;
   REC^LEN := 68;
   BLOCK^LEN := 4096;
   KEY^LEN := 68;
   KEY^OFFSET := 0;

   ERROR := FILE_CREATE_(KEYSFILE:ZSYS^VAL^LEN^FILENAME,
                         LENGTH,
                         !file^code!,
                         !primary^extent^size!,
                         !secondary^extent^size!,
                         !max^extents!,
                         !file^type!,
                         !options!,
                         REC^LEN,
                         BLOCK^LEN,
                         KEY^LEN,
                         KEY^OFFSET);

!  Create the alternate-key file for ALT3:

   KEYSFILE ':=' "$ADMIN.OPERATOR.ALT3" -> @S^PTR;
   LENGTH := @S^PTR '-' @KEYSFILE;
   REC^LEN := 68;
   BLOCK^LEN := 4096;
   KEY^LEN := 68;
   KEY^OFFSET := 0;
   ERROR := FILE_CREATE_(KEYSFILE:ZSYS^VAL^LEN^FILENAME,
                         LENGTH,
                         !file^code!,
                         !primary^extent^size!,
                         !secondary^extent^size!,
                         !max^extents!,
                         !file^type!,
                         !options!,
                         REC^LEN,
                         BLOCK^LEN,
                         KEY^LEN,
                         KEY^OFFSET);
END;
```

Adding Keys to an Alternate-Key File

Usually, you do not add keys to an alternate-key file directly. The file system inserts the alternate keys automatically whenever a new key is added to the primary-key file.

However, you can create an alternate-key file and specify that updates will not be done automatically. For example, in an application where a specific alternate key will not be used until some time after the primary file is updated, your application can choose to batch updates to an alternate-key file and then have the updates performed later. Such an approach means that you have to access the alternate-key file directly.

If you do need to access the alternate-key file directly, then the file system is unable to provide the same protection as when you access alternate keys using the file number of the primary file. Here, you must protect your alternate-key files from duplicate insertions when concurrent insertions take place on the primary file. To do this, you must set the alternate-key insertion locking mode, using SETMODE function 149, as follows:

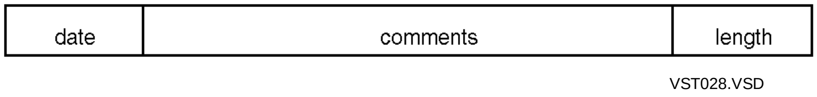
```
LITERAL ALT^KEY^INSERTION^LOCKING = 149,
        AUTO^LOCK                 = 1;
  .
  .
CALL SETMODE(F^NUM,
             ALT^KEY^INERTION^LOCKING,
             AUTO^LOCK);
```

This procedure call provides record-level locks while a record is being inserted in the primary data file. The lock is released as soon as the insert is complete.

Using Alternate Keys With a Relative File

Alternate keys used with a relative file reference a record number. As with any alternate-key mechanism, each occurrence of an alternate key references a primary key. Recall that for relative files, the primary key is the record number.

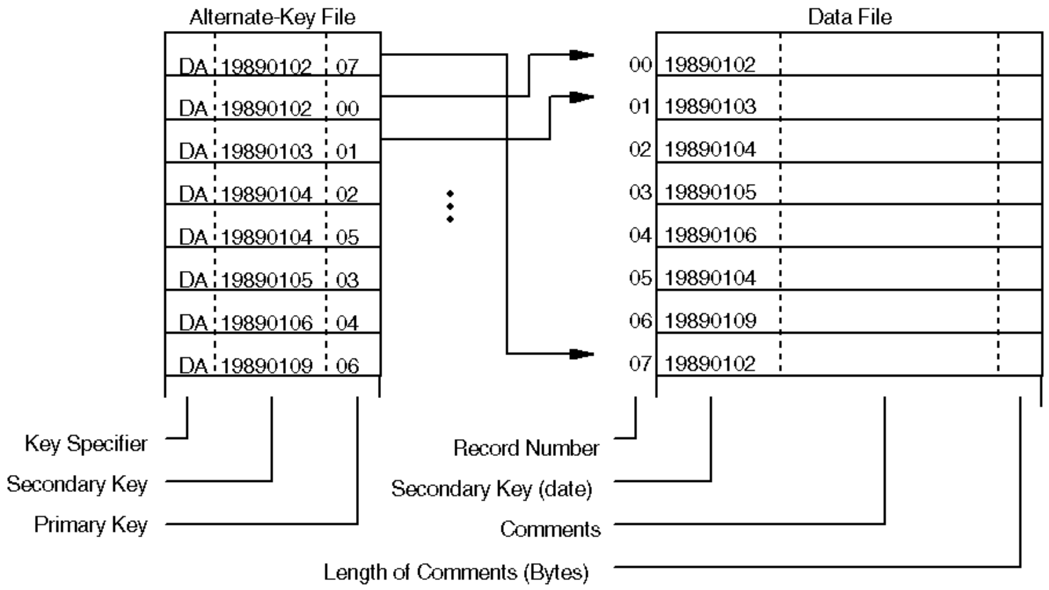
The log-file programming example from the subsection Using Relative Files will be enhanced to show how alternate keys can be used with relative files. The old example used a record to contain comments entered by the user. Here, two fields are added to the record structure so that one field can contain the user's comments, on the date, and a third field contains the length of the comments in bytes:


date
comments
length

VST028.VSD

The date field serves as the alternate key, enabling the user to look up information in the log using the date. “DA” will be used for the key specifier, short for “date.” The alternate-key and primary-key files look something like the example shown in Figure 5-9.

Figure 5-9. Example of Alternate-Key File for Use With a Relative File


Alternate-Key File
DA 19890102 07
DA 19890102 00
DA 19890103 01
DA 19890104 02
DA 19890104 05
DA 19890105 03
DA 19890106 04
DA 19890109 06
Data File
00 19890102
01 19890103
02 19890104
03 19890105
04 19890106
05 19890104
06 19890109
07 19890102
Key Specifier
Secondary Key
Primary Key
Record Number
Secondary Key (date)
Comments
Length of Comments (Bytes)

VST029.VSD

Alternate keys suit this application because:

- Using a key value such as the date is a convenient way of accessing data. (The relative-file example shown earlier in this section expects the user to know the record number.)
- The user can make more than one entry per day in the log, because alternate keys can be duplicated.
- Log entries can be made in any order, because the user can read sequentially by alternate-key value instead of by physical record sequence (as you would get when reading by record number).

The sample program needs a new primary-key file because of the new record structure. You also need an alternate-key file. You can create these files programmatically using the FILE_CREATE[LIST]_ procedure as described under Creating Alternate-Key Files earlier in this section, or you can use the FUP utility.

The following example uses FUP to create a primary-key file called ALTLOG and an alternate-key file called ALTKEY:

```
1> FUP
-SET TYPE R
-SET BLOCK 2048
-SET REC 512
-SET ALTKEY ("DA",KEYOFF 0,KEYLEN 8)
-SET ALTFILE (0,ALTKEY)
-SHOW
    TYPE R
    EXT ( 1 PAGES, 1 PAGES )
    REC 512
    BLOCK  2048
    ALTKEY ( "DA", FILE 0, KEYOFF 0, KEYLEN 8 )
    ALTFILE ( 0, $ADMIN.OPERATOR.ALTKEY)
    ALTCREATE
    MAXEXTENTS 16
-CREATE $ADMIN.OPERATOR.ALTLOG
CREATED - $ADMIN.OPERATOR.ALTLOG
CREATED - $ADMIN.OPERATOR.ALTKEY
-EXIT
2>
```

The sample program shown in this subsection enhances the program given in the subsection Using Relative Files to use alternate keys. The major changes are summarized as follows:

- The program declares a data structure RECORD to describe each record. Each record contains a 8-character date (in the format *yyyymmdd*), a 502-character field for the user’s comments, and an integer value representing the length of the comments in bytes. The READ^RECORD, UPDATE^RECORD, and INSERT^RECORD procedures all use this data structure when reading records from the data file or writing records to the data file.

- The GET^DATE procedure has been added to prompt the user for the date and check its length.
- The READ^RECORD procedure is modified as follows:
  - The procedure prompts the user for a date instead of a record number.
  - The procedure positions the pointers using the KEYPOSITION procedure (instead of POSITION). KEYPOSITION uses the alternate key to position the pointers. It also uses approximate positioning mode to accept a key value that does not exist. This is a useful feature, for example, if the user wants to read the log for the month of April and there was no entry for April 1.
  - Positioning by alternate key also causes sequential reads to be done in alternate-key sequence rather than by record number.
- The UPDATE^RECORD procedure also prompts the user for a date and then positions the pointers using this date and the KEYPOSITION procedure. Here, the positioning mode is exact. If there is no such key, then the procedure displays “No such record” and returns to the LOGGER procedure.
- The INSERT^RECORD procedure replaces the APPEND^RECORD procedure. It works like APPEND^RECORD, except that it prompts the user separately for the date and comments. Note that positioning is still done using the POSITION procedure, because the program simply adds records to the end of the file.
- The LOGGER procedure does not change. The alternate-key file is automatically opened when the corresponding data file is opened. Therefore there is no need for a separate open.

The following complete program applies alternate keys to a relative file.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !maximum file-name
                                        ! length
LITERAL DATESIZE = 8;            !size of date field
LITERAL COMMENTSIZE = 502;       !size of comment field

STRING .S^PTR;                   !pointer to end of string
INT     LOGNUM;                  !log file number
INT     TERMNUM;                 !terminal file number

STRUCT .RECORD;
BEGIN
   STRING DATE[0:DATESIZE-1];
   INT    DATA^LEN;
   STRING DATA[0:COMMENTSIZE-1];
END;

LITERAL BUFSIZE = COMMENTSIZE;
STRING .SBUFFER[0:BUFSIZE];      !terminal I/O buffer
                                 ! (one extra character)

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,
?  PROCESS_STOP_,READX,KEYPOSITION,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN,POSITION)
?LIST
```

```
!------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and print
! messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT(N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

    DEFINE  PRINT^STR(S) =  BEGIN   START^LINE;
                                    PUT^STR (S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is used when the
! file is not open, when there is no file number for it.
! FILE^ERRORS is used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN - 1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, the procedure
! attempts to display a message about the error and stop
! the program.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to ask the user for the next function to do:
!
! "r" to read records
! "u" to update a record
! "i" to insert a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'u' for Update Log, ");
   PRINT^STR("     'i' for Insert a comment, ");
   PRINT^STR("     'x' for Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!-------------------------------------------------------------
! Procedure for getting a date from the user.  The date
! entered is returned in SBUFFER.
!-------------------------------------------------------------

PROC GET^DATE;
BEGIN
   INT         COUNT^READ;

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Date (yyyymmdd):  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
   IF COUNT^READ <> DATESIZE THEN
   BEGIN
      START^LINE;
      PUT^STR("The date should be ");
      PUT^INT(DATESIZE);
      PUT^STR(" characters.");
      PRINT^LINE;
      GOTO PROMPT^AGAIN;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r."  The start of the read is selected randomly by record
! number.  The user has the option of sequentially reading
! subsequent messages.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   LITERAL     APPROX = 0;
   INT         KEY^SPEC;
   INT         COUNT^READ;
   INT         STATUS;
   INT         ERROR;

!  Prompt the user to select a record by entering a date:

   CALL GET^DATE;

!  Position the current-record and next-record pointers to
!  the selected record:

   KEY^SPEC ':=' "DA";
   CALL KEYPOSITION(LOGNUM,SBUFFER,KEY^SPEC,
                    !length^word!,
                    APPROX);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

!  Loop, reading and displaying records, until the user
!  declines to read the next record (any response other than
!  "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Read a record from the log file.  If the end of file is
   !  reached, return control to the LOGGER procedure:

      CALL READX(LOGNUM,RECORD,BUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(LOGNUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(LOGNUM);
      END;
   !  Print the record on the terminal:

      PRINT^STR("Date:      " & RECORD.DATE FOR DATESIZE);
      PRINT^STR("Comments:  " & RECORD.DATA FOR
                                RECORD.DATA^LEN);

      PRINT^BLANK;

   !  Prompt the user to read the next record.  The user
   !  must respond "y" to accept, otherwise the procedure
   !  returns to select the next function:

      SBUFFER ':=' ["Do you want to read another ",
                                 "record (y/n)? "]
                    -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                     BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

      SBUFFER[COUNT^READ] := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

```
!------------------------------------------------------------
! Procedure for updating a record.  The user selected
! function "u."  The user is prompted for the key of the
! record to update.  The procedure displays the current
! contents and prompts for the new.  After the user enters
! the new contents, the procedure updates the log file.
!------------------------------------------------------------

PROC UPDATE^RECORD;
BEGIN
   LITERAL     EXACT = 2;
   INT         KEY^SPEC;
   INT         COUNT^READ;
   INT         STATUS;
   INT         ERROR;

!  Prompt the user to select a record:

   CALL GET^DATE;

!  Position the current-record and next-record pointers to
!  the selected record:

   KEY^SPEC ':=' "DA";
   CALL KEYPOSITION(LOGNUM,SBUFFER,KEY^SPEC,
                    !length^word!,
                    EXACT);
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

!  Read the record.  Return to LOGGER if the record does not
!  exist:

   CALL READX(LOGNUM,RECORD,$LEN(RECORD),COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(LOGNUM,ERROR);
      IF (ERROR = 1) OR (ERROR = 11) THEN
      BEGIN
         PRINT^STR("No such record");
         RETURN;
      END;
      CALL FILE^ERRORS(LOGNUM);
   END;

!  Write the record to the terminal screen:

   PRINT^BLANK;
   PRINT^STR("Date:      " & RECORD.DATE FOR DATESIZE);
   PRINT^STR("Comments:  " & RECORD.DATA FOR
                             RECORD.DATA^LEN);

!  Prompt the user for the updated record:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Revised Comments:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   COMMENTSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   RECORD.DATA ':=' SBUFFER FOR COUNT^READ;
   RECORD.DATA^LEN := COUNT^READ;

!  Write new record to log file:

   CALL WRITEUPDATEX(LOGNUM,RECORD,$LEN(RECORD));
   IF <> THEN CALL FILE^ERRORS(LOGNUM);

END;
```

6

Communicating With Processes

This section describes how to use file-system procedures to communicate with other processes. Specifically, this section covers the following topics:

- How processes engage in two-way communication. Here, a process sends a message to another process. After processing the message, the recipient replies to the message.
- How processes engage in one-way communication. In one-way communication, the sender receives no meaningful information from the recipient. However, there is a variation on one-way communication where, although the sender receives no meaningful data in the reply, it does receive an error code.
- How a server processes messages concurrently, then replies to them in any order.
- How to handle system messages.

At the end of the section is a complete example of a simple application that makes use of requesters and servers. For complex examples, see Section 21, Writing a Requester Program, and Section 22, Writing a Server Program.

Throughout this section it is assumed that all processes involved already exist. Section 2, Using the File System, gives some examples of how to create processes. For more details about processes in general, refer to Section 16, Creating and Managing Processes.

This section does not describe how to process the Startup message; Section 8, Communicating With a TACL Process, provides details. Nor does this section describe how user processes pass the Startup message to each other; Section 16, Creating and Managing Processes, provides details. For details of a simplified process start-up using SIO procedures, refer to Section 15, Using the Sequential Input/Output Procedures.

This section does not discuss the use of sync IDs. It is possible, following a failure, that a process could receive the same message twice. Sync IDs are used to determine which one of a duplicated set of messages a process should respond to following a failure.

Sending and Receiving Messages: An Introduction

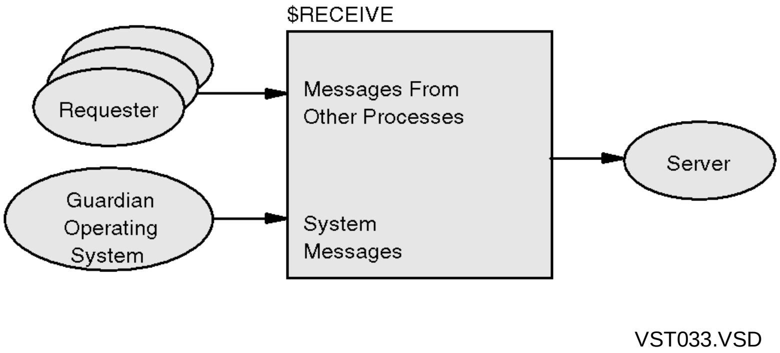
Interprocess communication (IPC):

- Permits a user process to receive messages from other user processes, thus providing the basis of the requester/server model introduced in Section 1, Introduction to Guardian Programming, as well as allowing processes to pass information to child processes by way of a Startup message (see Section 8, Communicating With a TACL Process).

- Permits user processes to receive system messages.

A process sends a message to another process by opening the recipient process file and writing a message to it. A process receives a message—whether the message is a request from another user process or a system message—by reading from a special file called $RECEIVE. Figure 6-1 shows these concepts.

Figure 6-1. Sending and Receiving Messages



Communication with other processes is done using procedure calls to the file system. The relevant procedures are introduced below:

AWAITIO[X]	Checks for completion of I/O for any system procedure. Specific to process communication, it checks for completion of read operations pending on the $RECEIVE file; AWAITIO checks READ and READUPDATE operations, and AWAITIOX checks READX and READUPDATEX operations as well as for READ and READUPDATE.
CANCEL	Cancels the oldest outstanding operation on a process or $RECEIVE file.
CANCELREQ	Cancels a message identified by a tag value.
CONTROL	Issues CONTROL operations to a process that simulates an I/O device.
CONTROLBUF	Issues CONTROLBUF operations to a process that simulates an I/O device.
FILE_CLOSE_	Terminates access to a process file or to the $RECEIVE file.

FILE_GETINFO_	Provides error information and characteristics about the open process file or $RECEIVE file.
FILE_GETRECEIVEINFO_	Returns information about the last message read from the $RECEIVE file. The information includes a tag that identifies the message.
FILE_OPEN_	Establishes communication with a process file for sending messages or with the $RECEIVE file for receiving messages.
MESSAGESTATUS	Checks the $RECEIVE file to see whether a specific message has been canceled.
READ[X]	Reads information from the $RECEIVE file. The READ procedure reads data into a buffer in the user data segment of the reading process. The READX procedure reads data into a buffer in either the user data segment or an extended data segment of the reading process.
READUPDATE[X]	Reads a message from $RECEIVE, expecting to reply to the message sender in a subsequent call to the REPLY procedure. The READUPDATE procedure reads data into a buffer in the user data segment of the receiving procedure. The READUPDATEX procedure reads data into a buffer in either the user data segment or an extended data segment of the receiving process.
REPLY[X]	Replies through the $RECEIVE file to a message that was previously read by READUPDATE. Optionally, REPLY[X] uses the message tag returned from FILE_GETRECEIVEINFO_ to designate which message is replied to. The REPLY procedure returns data from a buffer in the user data segment of the replying process. The REPLYX procedure returns data from a buffer either in the user data segment or in an extended data segment of the replying process.
SETMODE	Issues SETMODE functions to a process that simulates an I/O device. SETMODE is also used to turn on/off message queuing by priority of the sending process and to check whether any message has been canceled.
SETMODENOWAIT	Performs the same functions as SETMODE but in a nowait manner.
SETPARAM	Issues SETPARAM operations to a process that simulates an I/O device.

WRITE[X]	Sends a message to another process and waits for a reply (assuming waited I/O). WRITE[X] ignores the reply data and is often used to send a server request for which the server does not send any reply data. The WRITE procedure sends data from a buffer in the user data segment of the sending process. The WRITEX procedure sends data from either the user data segment or an extended data segment of the sending process.
WRITEREAD[X]	Sends a message to another process and waits for a reply from that process. The WRITEREAD procedure sends data from a buffer in the user data segment of the sending process. The WRITEREADX procedure sends data from either a buffer in the user data segment or a buffer in an extended data segment of the sending process.

For details about each of the above procedures, refer to the *Guardian Procedure Calls Reference Manual*.

Sending Messages to Other Processes

A requester process initiates communication with a server process by sending a message (a request) to the server. To do so, the requester process typically executes the following sequence:

1. Open the server process.
2. Create the request in a buffer in the requester's user data stack or extended data segment.
3. Send the message to the server, optionally waiting for a reply.

The following paragraphs explain how to do this.

Opening a Process

You open a process by passing the process file name as the *`file-name`* parameter to the FILE_OPEN_ procedure. The process descriptor can be named or unnamed, as described in Section 2, Using the File System.

Refer to Section 2, Using the File System, for information on how to create a process. For a thorough discussion of processes, refer to Section 16, Creating and Managing Processes.

Examples of Opening a Process

The following example opens process $SER1 for waited I/O.

```
FILE^NAME := "$SER1" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    PROC^NUM);
IF ERROR <> 0 THEN ...
```

Alternatively, you can open the server file for nowait I/O:

```
NOWAIT^DEPTH := 1;
FILE^NAME ':=' "$SER1" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    PROC^NUM,
                    !access!,
                    !exclusion!,
                    NOWAIT^DEPTH);
IF ERROR <> 0 THEN ...
```

The use of waited or nowait I/O affects the way you send messages to the server process. Writing Messages to Another Process, later in this section, explains this.

When Does the Open Finish?

The sample code fragments for opening a process work for opening any server process. However, the time at which the open finishes depends on the way the server process opens $RECEIVE.

If the server has not yet opened $RECEIVE, the requester’s open will not finish until it does. Once the server opens $RECEIVE, the open finishes at one of three points in time:

- If the server opens $RECEIVE without requesting system messages, the requester’s open finishes as soon as the server has opened $RECEIVE.
- If the server opens $RECEIVE to request system messages and to enable two-way communication (*`receive-depth`* parameter set to a value greater than zero), the requester’s open finishes when the server replies to the Open message.
- If the server opens $RECEIVE to request system messages but does not enable two-way communication (*`receive-depth`* parameter set to zero), the requester’s open finishes when the server reads the Open message.

See Receiving and Processing System Messages, later in this section, for information about opening $RECEIVE to receive system messages.

See Opening $RECEIVE for Two-Way Communication and Opening $RECEIVE for One-Way Communication later in this section for details about setting the *`receive-depth`* parameter.

Writing Messages to Another Process

Once the process file is open, you can communicate with the process by writing a message to the file number returned by the FILE_OPEN_ call. To send a message, you use either the WRITE[X] or WRITEREAD[X] procedure. For two-way communication (reply data expected), you should use WRITEREAD[X]. For one-way communication (no reply data expected) or one-way communication with error return, you can use a call to WRITE[X] or WRITEREAD[X].

As mentioned earlier under Opening a Process, the requester can open the server process for waited or nowait I/O. If the requester can wait for a reply, it should use waited I/O. If the requester cannot wait, it should initiate communication using nowait I/O and complete the communication later with a call to the AWAITIO[X] procedure. This is an application design issue. Refer to Section 4, Using Nowait Input/Output, for a detailed discussion of nowait I/O.

Writing a Message: No Reply Data Expected

The following example writes a message to a process without expecting any reply data. Here, the process has been opened using waited I/O.

```
LENGTH := $LEN(REQUEST.MESSAGE);
SBUFFER ':=' REQUEST.MESSAGE FOR LENGTH;
CALL WRITEX(PROC^NUM,
            SBUFFER,
            LENGTH,
            COUNT^WRITTEN);
IF <> THEN ...
```

The WRITEX procedure returns when the recipient process has read the message by issuing a READ[X] procedure call or has called REPLY[X] to respond to the message after having read it with READUPDATE[X]; that is, the sender and recipient processes remain synchronized. The `count-written` parameter shows how many bytes were read by the recipient process.

If you use WRITEREAD[X] to send a one-way message, then that call also returns as soon as the recipient process issues a READ[X] procedure call (or a READUPDATE[X] procedure call followed by a REPLY[X] call). In this case, the WRITEREAD[X] procedure returns no bytes.

If you opened the process using nowait I/O, then the WRITE[X] procedure returns immediately. The requester and server become synchronized when the requester completes the corresponding call to AWAITIO[X]. The following example shows how this part of the requester might be coded:

```
LENGTH := $LEN(REQUEST.MESSAGE);
SBUFFER ':=' REQUEST.MESSAGE FOR LENGTH;
CALL WRITEX(PROC^NUM,
            SBUFFER,
            LENGTH,
            COUNT^WRITTEN);
IF <> THEN ...
```

```
  .
  .
CALL AWAITIOX(PROC^NUM);
IF <> THEN ...
```

Writing a message for one-way communication with error return is no different from that given in the above examples. The only difference is in the server process, which must use READUPDATE[X] and REPLY[X].

Writing a Message: Reply Data Expected

Two-way communication expects reply data in reply to a written message. Here, you use the WRITEREAD[X] procedure to send the message to the server and receive the reply from the server in the same buffer.

Note that it is the action taken by the server process that determines whether one-way or two-way communication is being used. For two-way communication, the recipient process reads the message using a READUPDATE[X] procedure and then replies to the message using the REPLY[X] procedure. WRITEREAD[X] returns when the REPLY[X] procedure finishes, keeping the processes synchronized.

The following example sends a request for a database access to a server process. The reply returns the information retrieved from the database. This example assumes waited I/O.

```
STRUCT .RECORD;
BEGIN
   INT FUNCTION^CODE;
   INT ACCOUNT^NUMBER;
   INT AMOUNT;
END;

RECORD.FUNCTION^CODE := ADD;
RECORD.AMOUNT := 250;
RECORD.ACCOUNT^NUMBER := 16735;
WCOUNT := $LEN(RECORD);
RCOUNT := $LEN(RECORD);
CALL WRITEREADX(PROC^NUM,
                RECORD,
                WCOUNT,
                RCOUNT,
                COUNT^READ);
IF <> THEN ...
```

Had the process been opened for nowait I/O, the WRITEREAD[X] procedure would return immediately. The associated AWAITIO[X] call would finish on receipt of the reply and synchronize the processes.

Note. It is possible for the server process to send reply data even though the requester used the WRITE[X] procedure instead of WRITEREAD[X]. The file system simply discards the reply without even sending an error code to the requester or the server.

Queuing Messages on $RECEIVE

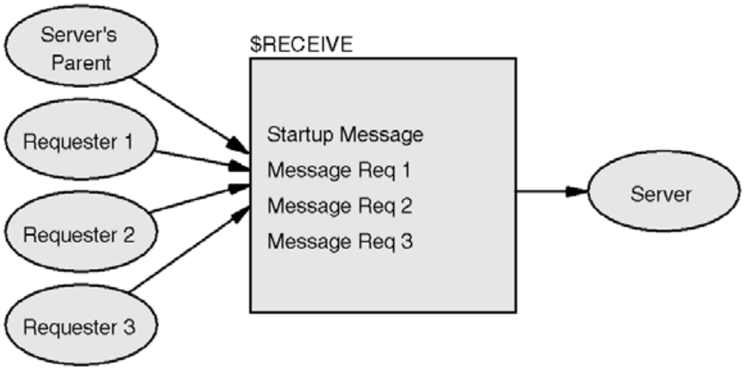
Messages destined for a given server process are placed by the file-system software in a queue in the $RECEIVE file for the process in question. Note that this queue exists in main memory, not on disk.

Normally, the queue is organized so that the server process reads messages from $RECEIVE in the order in which they arrive. This is usually true whether messages came from application requester processes, the server's parent process, or the operating system.

Note. Some system status messages (such as message -2, the CPU Down message) do get delivered ahead of messages from processes, including processes that are part of the operating system.

Figure 6-2 shows a typical queue. In this example, the server process has received messages from its parent process (the Startup message) and messages from each of three requester processes.

Figure 6-2. Multiple Requester Processes and Message Queuing



VST034.VSD

As an alternative to reading messages from $RECEIVE in the order in which they arrive, you can have the queue reordered according to the priority of the messages. Usually, the priority of a message is the same as the priority of the process that sent the message.

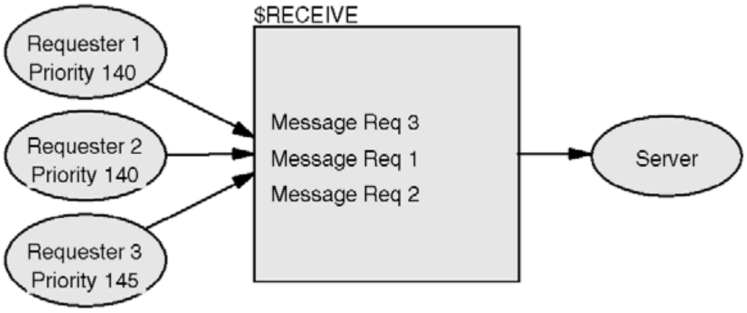
To reorder the message queue according to message priority, the server process must issue SETMODE function 36 as follows:

```
LITERAL PRIORITY^QUEUING = 36,
        ON               = 1,
        OFF              = 0;
  .
  .
CALL SETMODE(RECV^NUM,
             PRIORITY^QUEUING,
             ON);
IF <> THEN ...
```

The priority of every process is set when the process is created. Refer to Section 16, Creating and Managing Processes, for details of how to do this using the PROCESS_CREATE_ procedure.

Figure 6-3 shows how messages are queued according to sender process priority.

Figure 6-3. Message Queuing by Process Priority


$RECEIVE
Requester 1
Priority 140
Requester 2
Priority 140
Requester 3
Priority 145
Message Req 3
Message Req 1
Message Req 2
Server

VST035.VSD

Receiving and Replying to Messages From Other Processes

First look at how two-way communication works. This subsection is concerned with processes that read a message, process the message, and then reply to the sender before reading the next message. The file system keeps track of where to send replies. The following paragraphs describe how to perform two-way communication.

Here, it is assumed that the server processes each message in turn. That is, the server reads a message from the top of $RECEIVE, processes and replies to this message, and then reads the next message.

It is possible to read several messages and then process them in any order. Doing so involves putting each message on a list of messages that have been read but not replied to and then removing the message from the queue when the message is replied to. The subsection Handling Multiple Messages Concurrently, later in this section, describes how to do this.

Opening $RECEIVE for Two-Way Communication

The receiving process reads messages from the $RECEIVE file. For two-way communication, the process must set the *receive-depth* to a value greater than zero.

The receive depth specifies how many messages can be read by the server process before any message is replied to. For one-way communication, this value defaults to zero because no reply is intended and therefore there is no need to queue messages in this way. When processing and replying to one message at a time, however, the maximum number of messages that can be read but not replied to is one. Hence the server process is opened with a receive depth of 1.

You set the receive depth using a parameter of the FILE_OPEN_ procedure as shown below:

```
FILE^NAME ':=' "$RECEIVE";
LENGTH := 8;
RECV^DEPTH := 1;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    RECV^NUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    RECV^DEPTH);
IF ERROR <> 0 THEN ...
```

Note that although the *receive-depth* parameter is in the same position in the procedure call as the *sync-depth* parameter for disk-file opens, the purpose of the parameter is different.

Reading Messages for Two-Way Communication

Use the READUPDATE[X] procedure to read a message from $RECEIVE if you want to reply to the message with data. READUPDATE[X] reads the message without terminating the WRITEREAD[X] procedure issued by the sender of the message. The WRITEREAD[X] procedure instead waits for a reply.

An example of a READUPDATEX call follows:

```
CALL READUPDATEX(RECV^NUM,
                 SBUFFER,
                 RCOUNT);
IF <> THEN ...
```

Replying to Messages

After reading a message from $RECEIVE using the READUPDATEX procedure, use the REPLYX procedure to send the reply. This procedure sends data back to the sender of the original message and returns an error indication:

```
STRUCT .RECORD;
BEGIN
   INT FUNCTION^CODE;
   INT ACCOUNT^NUMBER;
   INT AMOUNT;
END;

RECORD.AMOUNT := 1250;
WCOUNT := $LEN(RECORD);
ERROR := 0;
CALL REPLYX(RECORD,
            WCOUNT,
            !count^written!,
            !message^tag!,
            ERROR);
IF <> THEN ...
```

Returning Data

In the example above, the first parameter in the REPLYX call contains the reply message; in this case, a data record requested by the sender.

Returning Error Information

The ERROR parameter in the above example can also be returned to the message originator. Its purpose is to return an indication that there is a problem. Typically, the reply data does not contain the expected result. The requested operation may not have been completely and correctly performed.

The range of error codes between 256 and 511 is reserved for application programs to use. You can freely define the meanings of these error codes yourself, as part of the relationship between the requester and the server.

Error numbers outside the range 256 through 511 are reserved for the operating system and should not be used arbitrarily, because this could interfere with correct error handling inside the operating system and file system.

If there is no problem, a value of zero (the default value) will be returned.

The requester process should call FILE_GETINFO_ after returning from the WRITEREAD[X] call to obtain the returned error code, just as you would use FILE_GETINFO_ to obtain any file-system error code.

Sending, Receiving, and Replying to Messages: An Example

In the example shown in Figure 6-4, the server process sends a reply back to the requester. This example allows the user of the terminal running the requester process to query the user of the terminal running the server process. The purpose of the example is to show the concept. The long example at the end of this section shows a practical use.

The programs work like this: Initially, the requester prompts its terminal user for message input using WRITEREADX, and then it sends the message to the server. The server (which has opened the $RECEIVE file with a receive depth of 1) uses the READUPDATEX procedure to read the message from $RECEIVE so as not to terminate the requester’s WRITEREADX without reply data. The server displays the received message on its home terminal using WRITEREADX, which solicits a reply from the terminal user. The server process then returns the reply to the requester using the REPLYX procedure. Finally, the requester displays the reply on its home terminal and waits for further input from the user. Both processes terminate when the server user types “EXIT.”

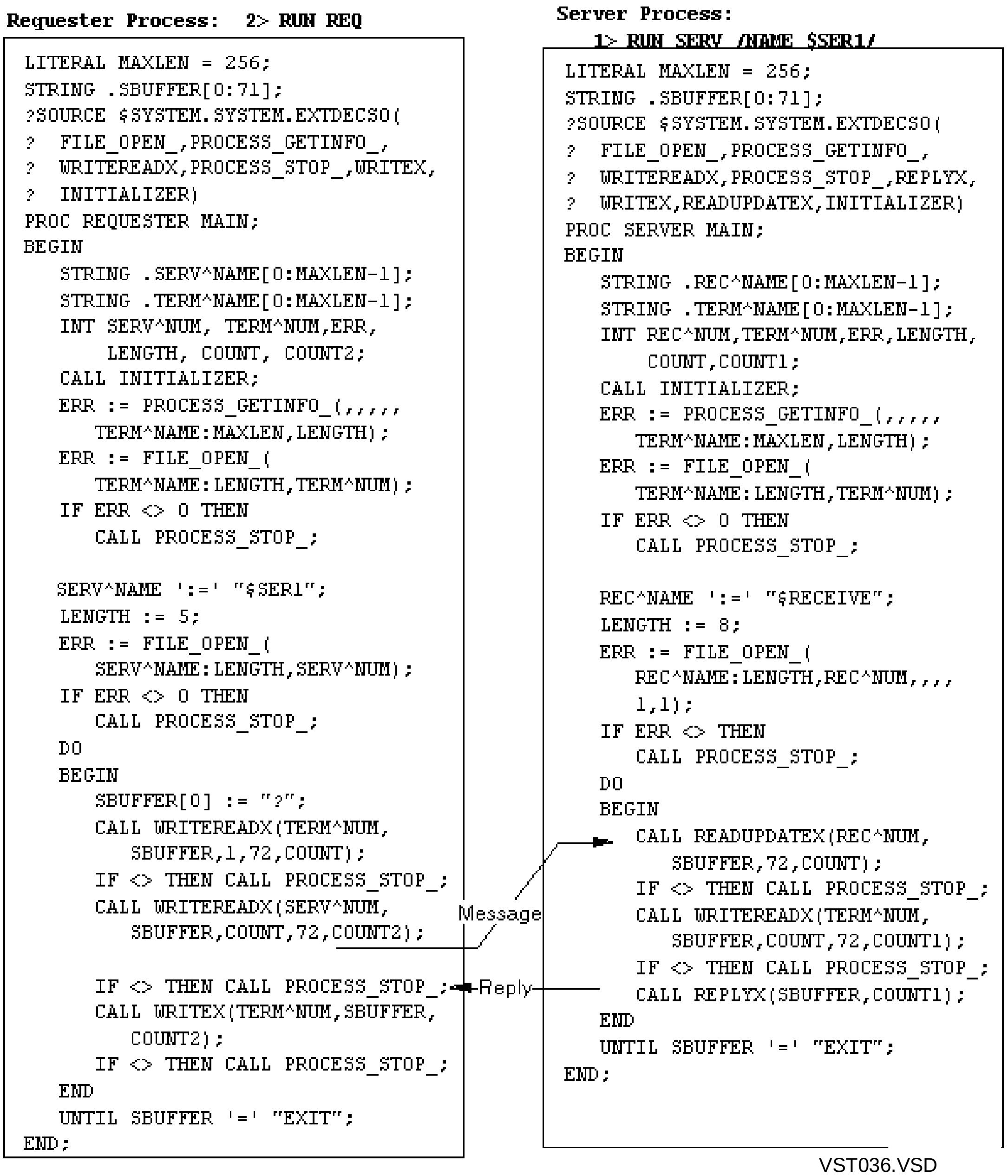
Closing $RECEIVE

You explicitly close $RECEIVE as you would any file using the FILE_CLOSE_ procedure:

```
ERROR := FILE_CLOSE_(RECV^NUM);
IF ERROR <> 0 THEN ...
```

As for any file, if you do not explicitly close $RECEIVE, then the file is implicitly closed when the process terminates.

Figure 6-4. Two-Way Interprocess Communication



Requester Process: 2> RUN REQ

```
LITERAL MAXLEN = 256;
STRING .SBUFFER[0:71];
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(
?  FILE_OPEN_,PROCESS_GETINFO_,
?  WRITEREADX,PROCESS_STOP_,WRITEX,
?  INITIALIZER)
PROC REQUESTER MAIN;
BEGIN
   STRING .SERV^NAME[0:MAXLEN-1];
   STRING .TERM^NAME[0:MAXLEN-1];
   INT SERV^NUM, TERM^NUM,ERR,
       LENGTH, COUNT, COUNT2;
   CALL INITIALIZER;
   ERR := PROCESS_GETINFO_(,,,,,
      TERM^NAME:MAXLEN,LENGTH);
   ERR := FILE_OPEN_(
      TERM^NAME:LENGTH,TERM^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;

   SERV^NAME ':=' "$SER1";
   LENGTH := 5;
   ERR := FILE_OPEN_(
      SERV^NAME:LENGTH,SERV^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;
   DO
   BEGIN
      SBUFFER[0] := "?";
      CALL WRITEREADX(TERM^NUM,
         SBUFFER,1,72,COUNT);
      IF <> THEN CALL PROCESS_STOP_;
      CALL WRITEREADX(SERV^NUM,
         SBUFFER,COUNT,72,COUNT2);

      IF <> THEN CALL PROCESS_STOP_;
      CALL WRITEX(TERM^NUM,SBUFFER,
         COUNT2);
      IF <> THEN CALL PROCESS_STOP_;
   END
   UNTIL SBUFFER '=' "EXIT";
END;
```

Server Process:
1> RUN SERV /NAME $SER1/

```
LITERAL MAXLEN = 256;
STRING .SBUFFER[0:71];
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(
?  FILE_OPEN_,PROCESS_GETINFO_,
?  WRITEREADX,PROCESS_STOP_,REPLYX,
?  WRITEX,READUPDATEX,INITIALIZER)
PROC SERVER MAIN;
BEGIN
   STRING .REC^NAME[0:MAXLEN-1];
   STRING .TERM^NAME[0:MAXLEN-1];
   INT REC^NUM,TERM^NUM,ERR,LENGTH,
       COUNT,COUNT1;
   CALL INITIALIZER;
   ERR := PROCESS_GETINFO_(,,,,,
      TERM^NAME:MAXLEN,LENGTH);
   ERR := FILE_OPEN_(
      TERM^NAME:LENGTH,TERM^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;

   REC^NAME ':=' "$RECEIVE";
   LENGTH := 8;
   ERR := FILE_OPEN_(
      REC^NAME:LENGTH,REC^NUM,,,
      1,1);
   IF ERR <> THEN
      CALL PROCESS_STOP_;
   DO
   BEGIN
      CALL READUPDATEX(REC^NUM,
         SBUFFER,72,COUNT);
      IF <> THEN CALL PROCESS_STOP_;
      CALL WRITEREADX(TERM^NUM,
         SBUFFER,COUNT,72,COUNT1);
      IF <> THEN CALL PROCESS_STOP_;
      CALL REPLYX(SBUFFER,COUNT1);
   END
   UNTIL SBUFFER '=' "EXIT";
END;
```

VST036.VSD

Receiving Messages From Other Processes: One-Way Communication

Now look at how one-way communication works. When receiving messages in one-way communication, all the server has to do is read the message and process it. No reply data is necessary in the reply.

To receive a message from another process, the server process must open the $RECEIVE file and read from it. The following paragraphs explain how to do this if your program will not send a reply to the sending process.

Opening $RECEIVE for One-Way Communication

As for two-way communication, the receiving process reads messages from the $RECEIVE file. For one-way communication, however, it is not necessary to set the *`receive-depth`* parameter to a nonzero value because the receiving process does not return any reply data. You therefore open the $RECEIVE file using the FILE_OPEN_ procedure as shown below:

```
FILE^NAME ':=' "$RECEIVE";
LENGTH := 8;
FILE_OPEN_(FILE^NAME:LENGTH,
           RECV^NUM);
IF <> THEN ...
```

Reading From $RECEIVE for One-Way Communication

Use the READ[X] procedure to read the first message from $RECEIVE. In addition to reading the message, the READ[X] procedure also keeps the requester and server processes synchronized by terminating the WRITE[X] or WRITEREAD[X] procedure call that the requester used to send the message.

```
CALL READX(RECV^NUM,
           SBUFFER,
           RCOUNT);
IF <> THEN ...
```

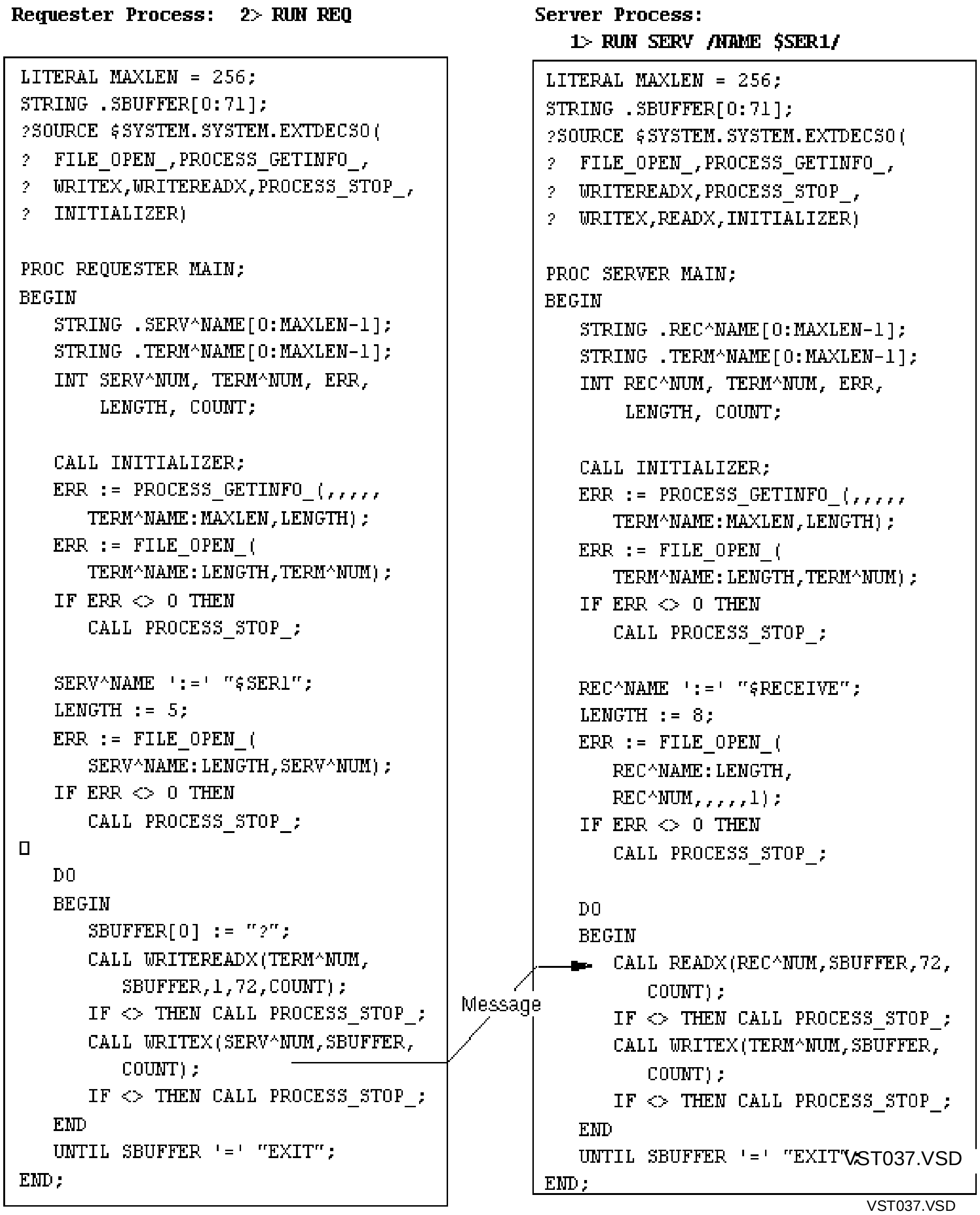
If you want to send an error response to the requester process without sending any other data, use the READUPDATE[X] and REPLY[X] procedures as used for two-way communication. You also need to open $RECEIVE with a receive depth of at least 1. Because the requester sent the message using the WRITE[X] procedure, WRITE[X] waits for the reply to finish but discards any data sent in the reply. By calling the FILE_GETINFO_ procedure after the WRITE[X] procedure, the requester can obtain the error code sent in the reply.

Sending and Receiving One-Way Messages: An Example

The example shown in Figure 6-5 shows one-way communication between a requester process and a server process. The requester process writes messages to the server process file as typed by the user at a terminal. The server reads each message from $RECEIVE and displays the message on the home terminal of the process. Both processes stop when the user at the terminal of the requester process types “EXIT.”

Again, the purpose of this example is to show the concept. The example does not necessarily perform a useful function.

Figure 6-5. One-Way Interprocess Communication



Requester Process: 2> RUN REQ

```
LITERAL MAXLEN = 256;
STRING .SBUFFER[0:71];
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(
?  FILE_OPEN_,PROCESS_GETINFO_,
?  WRITEX,WRITEREADX,PROCESS_STOP_,
?  INITIALIZER)

PROC REQUESTER MAIN;
BEGIN
   STRING .SERV^NAME[0:MAXLEN-1];
   STRING .TERM^NAME[0:MAXLEN-1];
   INT SERV^NUM, TERM^NUM, ERR,
       LENGTH, COUNT;

   CALL INITIALIZER;
   ERR := PROCESS_GETINFO_(,,,,,
      TERM^NAME:MAXLEN,LENGTH);
   ERR := FILE_OPEN_(
      TERM^NAME:LENGTH,TERM^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;

   SERV^NAME ':=' "$SER1";
   LENGTH := 5;
   ERR := FILE_OPEN_(
      SERV^NAME:LENGTH,SERV^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;
□
   DO
   BEGIN
      SBUFFER[0] := "?";
      CALL WRITEREADX(TERM^NUM,
         SBUFFER,1,72,COUNT);
      IF <> THEN CALL PROCESS_STOP_;
      CALL WRITEX(SERV^NUM,SBUFFER,
         COUNT);
      IF <> THEN CALL PROCESS_STOP_;
   END
   UNTIL SBUFFER '=' "EXIT";
END;
```

Server Process:
1> RUN SERV /NAME $SER1/

```
LITERAL MAXLEN = 256;
STRING .SBUFFER[0:71];
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(
?  FILE_OPEN_,PROCESS_GETINFO_,
?  WRITEREADX,PROCESS_STOP_,
?  WRITEX,READX,INITIALIZER)

PROC SERVER MAIN;
BEGIN
   STRING .REC^NAME[0:MAXLEN-1];
   STRING .TERM^NAME[0:MAXLEN-1];
   INT REC^NUM, TERM^NUM, ERR,
       LENGTH, COUNT;

   CALL INITIALIZER;
   ERR := PROCESS_GETINFO_(,,,,,
      TERM^NAME:MAXLEN,LENGTH);
   ERR := FILE_OPEN_(
      TERM^NAME:LENGTH,TERM^NUM);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;

   REC^NAME ':=' "$RECEIVE";
   LENGTH := 8;
   ERR := FILE_OPEN_(
      REC^NAME:LENGTH,
      REC^NUM,,,,,1);
   IF ERR <> 0 THEN
      CALL PROCESS_STOP_;

   DO
   BEGIN
      CALL READX(REC^NUM,SBUFFER,72,
         COUNT);
      IF <> THEN CALL PROCESS_STOP_;
      CALL WRITEX(TERM^NUM,SBUFFER,
         COUNT);
      IF <> THEN CALL PROCESS_STOP_;
   END
   UNTIL SBUFFER '=' "EXIT"VST037.VSD
END;
```

VST037.VSD

Handling Multiple Messages Concurrently

So far, you have seen how a server processes requests one at a time as it reads them from the $RECEIVE file. However, in some applications, it could happen that for the server to complete a request, it must wait for events outside the server process to finish. Other requests might have to wait a long time for the server to become available. By handling multiple requests concurrently, the server is able to process requests while waiting for longer-running requests to finish.

This subsection describes how the server can read several requests from $RECEIVE and then process and reply to them in any order. To do so, the server typically executes the following sequence:

1. Open $RECEIVE with a receive depth equal to the maximum number of requests to this process that you want to be able to process concurrently.
2. Read requests from $RECEIVE. The file system assigns a tag value to each message and keeps a list of all messages that you have read from $RECEIVE but not yet replied to.
3. Process these requests in any order. This gives the server the flexibility of assigning priority to requests or processing requests concurrently.
4. Reply to each message after processing. The file system removes the message from the list of messages that have not been replied to.

The following paragraphs describe how the server process performs these functions, including how to ensure, when you send a reply, that the reply goes to the process that issued the corresponding request.

Opening $RECEIVE to Allow Concurrent Message Processing

To open the $RECEIVE file and enable concurrent message processing, you need to set the *receive-depth* parameter equal to the maximum number of messages that your server program will queue before replying. The length of this list is an application design issue.

The following example sets the receive depth to 4:

```
FILE^NAME ':=' "$RECEIVE";
LENGTH := 8;
RECV^DEPTH := 4;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    RECV^NUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    RECV^DEPTH);
IF ERROR <> 0 THEN ...
```

Reading Messages for Concurrent Processing

You use the READUPDATE[X] procedure to read each message without terminating the corresponding WRITEREAD[X] call. The WRITEREAD[X] procedure finishes when you reply to the message using the REPLY[X] procedure. With queued messages, however, you should use message tags to make sure that each reply goes to the process that sent the message you are replying to.

When processing several messages concurrently, there needs to be a way to identify each message. The message tag returned by the FILE_GETRECEIVEINFO_ procedure can be used for this purpose. Remember that FILE_GETRECEIVEINFO_ gets information about the most recently read message. You therefore need to issue a call to this procedure following each READUPDATE call; for example:

```
CALL READUPDATEX(RECV^NUM,
                 SBUFFER0,
                 RCOUNT);
IF <> THEN ...;

ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;
TAG0 := INFORMATION[2];
 .
 .

CALL READUPDATEX(RECV^NUM,
                 SBUFFER1,
                 RCOUNT);
IF <> THEN ...;

ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;
TAG1 := INFORMATION[2];
 .
 .

CALL READUPDATEX(RECV^NUM,
                 SBUFFER2,
                 RCOUNT);
IF <> THEN ...;

ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;
TAG2 := INFORMATION[2];
 .
 .
```

Getting Information About Messages Read From $RECEIVE

In addition to the message tag, the FILE_GETRECEIVEINFO_ procedure returns additional information about the last message read from the $RECEIVE file. The FILE_GETRECEIVEINFO_ procedure returns information as follows:

```
STRUCT .INFORMATION(ZSYS^DDL^RECEIVEINFORMATION^DEF);
 .
 .
ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...
```

The returned information includes the following:

- The I/O operation issued by the sender
- The maximum length of the reply message
- The tag value that identifies the message
- The file number used by the sender for communicating with this process
- The sync ID for fault-tolerant processing
- The process handle that identifies the sending process
- The open label

This information is typically used by the server process to assign priorities for message handling and to establish message tracking.

This subsection discusses some of the more commonly used information. For further information, you should refer to Section 16, Creating and Managing Processes, for information on process handles or to the *Guardian Procedure Calls Reference Manual* for complete details about all information returned by the FILE_GETRECEIVEINFO_ procedure.

Getting the I/O Operation

The I/O operation specified by the requester is returned in word 0 of the value returned by the FILE_GETRECEIVEINFO_ procedure. It has one of the following values:

0	Indicates that the last message received was a system message. Refer to , later in this section, for a discussion of system messages.
1	Indicates that the last message received resulted from a WRITE[X] procedure call by the requester process.
2	Indicates that the last message received resulted from a READ[X] procedure call by the requester process.
3	Indicates that the requester process issued a WRITEREAD[X] procedure call.

The following example extracts the type of operation requested in the message:

```
STRUCT .INFORMATION(ZSYS^DDL^RECEIVEINFORMATION^DEF);
 .
 .
ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;

CASE INFORMATION.Z^IOTYPE OF
BEGIN
   ZSYS^VAL^RCV^IOTYPE^SYSTEMMSG ->!System message
   ZSYS^VAL^RCV^IOTYPE^WRITE     ->! Write request
   ZSYS^VAL^RCV^IOTYPE^READ      ->!READ[X] request
   ZSYS^VAL^RCV^IOTYPE^WRITEREAD ->!WRITEREAD[X] request
   OTHERWISE ->!Error
END;
```

Getting the Maximum Reply Count

The maximum reply count indicates the number of reply bytes expected by the sender. If the message received on $RECEIVE was generated by a WRITEREAD[X] procedure call, this value is the *`read-count`* value specified by the sender in the WRITEREAD[X] procedure call. If the sender issued a WRITE[X] call, then the maximum reply count is zero. The value can be nonzero for a READ[X] request or for a system message.

The value is returned by the FILE_GETRECEIVEINFO_ procedure in word 1. Your server process can use this value to ensure that the reply does not get truncated when read by the requester or to adjust a variable-length reply to the expected reply size.

The following example checks the size of the reply data and compares it with the expected reply size. If the reply data is larger than the expected reply size, then the server returns error number 300 to the requester to inform the requester that the data is truncated.

```
STRUCT .INFORMATION(ZSYS^DDL^RECEIVEINFORMATION^DEF);
 .
 .
!Get the expected reply length:
ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;


!Set error if reply longer than expected reply:
SBUFFER ':=' "Reply to Message" -> @S^PTR;
WCOUNT := @S^PTR '-' @SBUFFER;
IF WCOUNT > INFORMATION.Z^MAXREPLYCOUNT THEN ERROR := 300
   ELSE ERROR := 0;

!Reply to requester:
CALL REPLY(SBUFFER,
           WCOUNT,
           !count^written!,
           !message^tag!,
           ERROR);
IF <> THEN ...
```

Getting the Message Tag

The message tag identifies a message and is used when the recipient process may have to process multiple messages. The tag enables the recipient process to send the reply to the correct process, as described earlier in this section.

The message tag is returned by the FILE_GETRECEIVEINFO_ procedure in word 2:

```
STRUCT .INFORMATION(ZSYS^DDL^RECEIVEINFORMATION^DEF);
 .
 .
ERROR := FILE_GETRECEIVEINFO_(INFORMATION);
IF ERROR <> 0 THEN ...;
 .
 .

MESSAGE^TAG := INFORMATION.Z^MESSAGETAG;
CALL REPLY(BUFFER,
           WCOUNT,
           !count^written!,
           MESSAGE^TAG);
```

Replying to Messages

When replying to messages that were concurrently processed, you need to include the message tag as a parameter to the REPLY[X] procedure to ensure that the reply is sent to the correct process. The following example replies to the three messages received in the example under Reading Messages for Concurrent Processing, earlier in this subsection:

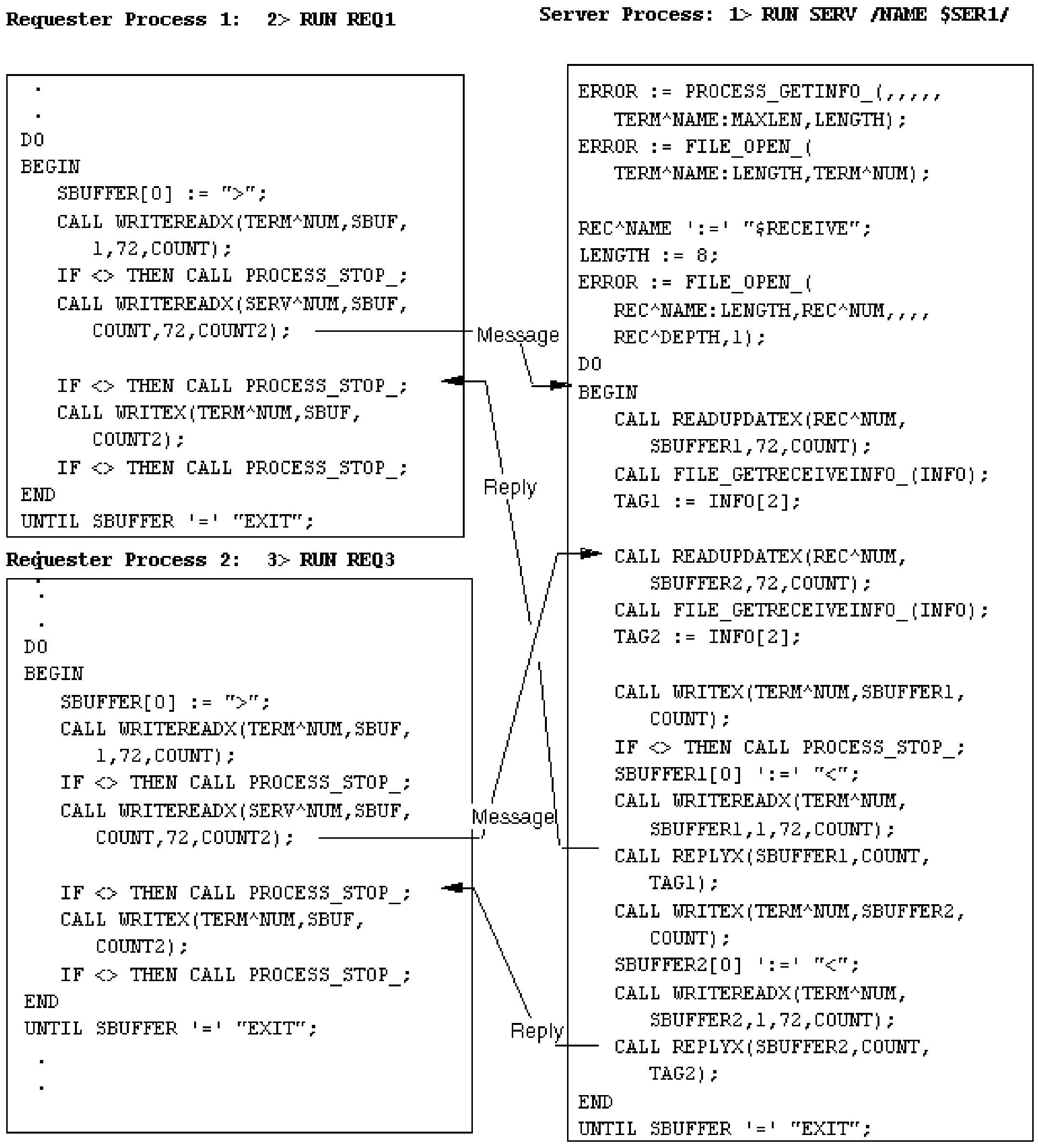
```
CALL REPLYX(SBUFFER0,
            WCOUNT,
            COUNT^WRITTEN,
            TAG0);
IF <> THEN ...;

CALL REPLYX(SBUFFER1,
            WCOUNT,
            COUNT^WRITTEN,
            TAG2);
IF <> THEN ...;

CALL REPLYX(BUFFER2,
            WCOUNT,
            COUNT^WRITTEN,
            TAG1);
IF <> THEN ...;
```

Note that the order of replying is different from the order of receiving.

Handling Multiple Messages Concurrently: An Example

Figure 6-6 shows an example of message queuing. It is similar to the example given in Figure 6-4, where concurrent message processing was not done. Here, however, the server process accepts input from two requesters, queues one message from each, and then processes and replies to both messages.

Figure 6-6. Example of Handling Multiple Messages Concurrently



Requester Process 1: 2> RUN REQ1

```
  .
  .
DO
BEGIN
   SBUFFER[0] := ">";
   CALL WRITEREADX(TERM^NUM,SBUF,
      1,72,COUNT);
   IF <> THEN CALL PROCESS_STOP_;
   CALL WRITEREADX(SERV^NUM,SBUF,
      COUNT,72,COUNT2);

   IF <> THEN CALL PROCESS_STOP_;
   CALL WRITEX(TERM^NUM,SBUF,
      COUNT2);
   IF <> THEN CALL PROCESS_STOP_;
END
UNTIL SBUFFER '=' "EXIT";
```

Requester Process 2: 3> RUN REQ3

```
  .
  .
DO
BEGIN
   SBUFFER[0] := ">";
   CALL WRITEREADX(TERM^NUM,SBUF,
      1,72,COUNT);
   IF <> THEN CALL PROCESS_STOP_;
   CALL WRITEREADX(SERV^NUM,SBUF,
      COUNT,72,COUNT2);

   IF <> THEN CALL PROCESS_STOP_;
   CALL WRITEX(TERM^NUM,SBUF,
      COUNT2);
   IF <> THEN CALL PROCESS_STOP_;
END
UNTIL SBUFFER '=' "EXIT";
  .
  .
```

Server Process: 1> RUN SERV /NAME $SER1/

```
ERROR := PROCESS_GETINFO_(,,,,,
   TERM^NAME:MAXLEN,LENGTH);
ERROR := FILE_OPEN_(
   TERM^NAME:LENGTH,TERM^NUM);

REC^NAME ':=' "$RECEIVE";
LENGTH := 8;
ERROR := FILE_OPEN_(
   REC^NAME:LENGTH,REC^NUM,,,,
   REC^DEPTH,1);
DO
BEGIN
   CALL READUPDATEX(REC^NUM,
      SBUFFER1,72,COUNT);
   CALL FILE_GETRECEIVEINFO_(INFO);
   TAG1 := INFO[2];

   CALL READUPDATEX(REC^NUM,
      SBUFFER2,72,COUNT);
   CALL FILE_GETRECEIVEINFO_(INFO);
   TAG2 := INFO[2];

   CALL WRITEX(TERM^NUM,SBUFFER1,
      COUNT);
   IF <> THEN CALL PROCESS_STOP_;
   SBUFFER1[0] ':=' "<";
   CALL WRITEREADX(TERM^NUM,
      SBUFFER1,1,72,COUNT);
   CALL REPLYX(SBUFFER1,COUNT,
      TAG1);
   CALL WRITEX(TERM^NUM,SBUFFER2,
      COUNT);
   SBUFFER2[0] ':=' "<";
   CALL WRITEREADX(TERM^NUM,
      SBUFFER2,1,72,COUNT);
   CALL REPLYX(SBUFFER2,COUNT,
      TAG2);
END
UNTIL SBUFFER '=' "EXIT";
```

VST038.VSD

Checking for Canceled Messages

Typically, a server processes messages from other processes as follows:

1. The server reads a message from its $RECEIVE file.
2. The server performs some processing in response to the message.
3. The server replies to the process that sent the message.

Between the time the server reads the message using the READUPDATE[X] procedure and the time the server replies to the message using the REPLY[X] procedure, the message could be canceled for any of the following reasons:

- The process that sent the message calls CANCEL, CANCELREQ, FILE_CLOSE_, or certain forms of AWAITIO[X].
- The process that sent the message stops executing (for example, by calling PROCESS_STOP_).

Note. It is never required to cancel a message. Only if a request takes a long time to process is it appropriate to cancel that request.

A typical use for message cancellation is when a requester process wants a high-priority request performed but the server is taking a long time to process a lower-priority request. The requester can ask to have the old request canceled so that the new request can proceed.

For example, a server process might wait indefinitely for input from a terminal or for a lock on a file. If the requester wants to have another request processed that conflicts with the long running request—like printing some text to the same terminal—then it can send a cancellation message to the server before sending the new request. Logic in the message system prevents the new request from overtaking the cancellation message and reaching the server first.

Even if a message that has been read using READUPDATE[X] is canceled, the server must still reply to that message by calling REPLY[X]. The response by the server is an application design issue that depends on the relationship between the requester and server. Typically, a requester that cancels a request expects that the original request may not be fully completed, therefore, the server need not perform any processing for that message. Moreover, a process that cancels a request and then sends a new request probably does not want the old request to hold up execution of the new request. Thus, the server process can avoid unnecessary processing by ensuring that a message has not been canceled before processing that message.

A process can check for canceled messages in two ways:

- By checking for system message number -38 (cancellation messages) in its $RECEIVE file. This method is appropriate only when the server handles multiple requests concurrently.
- By calling the MESSAGESTATUS procedure.

These methods are described in the following paragraphs. Both methods involve use of the message **tag**. The message tag is generally used to distinguish among multiple messages when the server chooses to concurrently process several messages (see Handling Multiple Messages Concurrently earlier in this section). Here, the message tag is used to identify the canceled message.

You use the FILE_GETRECEIVEINFO_ procedure to obtain the message tag of the last message read from $RECEIVE. This procedure returns information about the message just read.

```
STRUCT .INFORMATION(ZSYS^DDL^RECEIVEINFORMATION^DEF);
INT TAG;
 .
 .
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT);
IF <> THEN ...

CALL FILE_GETRECEIVEINFO_(INFORMATION);
TAG := INFORMATION.Z^MESSAGETAG;
```

Checking for Cancellation Messages

When a requester sends a message to a server and that message is later canceled (for any of the reasons stated earlier), the operating system sends a system message to the server to inform the server that the message has been canceled. The system message is called a cancellation message and is made up of two words: the first word contains the message type (-38), and the second word contains the message tag of the canceled message.

The effect of the message and the way the server process must respond are influenced by when the cancellation message arrives with respect to the processing of the request to be canceled:

- If the request has not yet been read by the server process, the operating system removes the request from $RECEIVE and the server never receives the cancellation message.
- If the request is currently being processed by the server, the cancellation message will be delivered. The server should stop processing the request. A REPLYX call is required. There is no point checking for a cancellation message immediately after reading the request because the requester will usually not issue a cancellation message right away.
- If the server has already replied to the request, the server will not receive the cancellation message.
- The cancellation message is not delivered if the server has read but not replied to the number of messages specified in the *`receive-depth`* parameter of the FILE_OPEN_ call.

If you check for cancellation messages, you should do so at points during message processing only if the processing takes a long time.

To enable receipt of cancellation messages in its $RECEIVE file, the server process must call SETMODE function 80. SETMODE function 80 controls several functions related to the $RECEIVE file, one of which is receipt of cancellation messages. For a list of the functions performed by SETMODE function 80, see the *Guardian Procedure Calls Reference Manual*.

The following example enables the receipt of cancellation messages. To receive cancellation messages, bit 13 of parameter 1 of the SETMODE function 80 call must be set to 1:

```
LITERAL SET^RECV^MSG^MODE = 80,
        ACCEPT^CANCEL^MESSAGES = %B0000000000000100;
  .
  .
CALL SETMODE(RECV^NUM,
             SET^RECV^MSG^MODE,
             ACCEPT^CANCEL^MESSAGES);
```

Once a call to SETMODE function 80 has been issued, you can check for cancellation messages as you would any other system message. See , later in this section.

Using the MESSAGESTATUS Procedure

Cancellation messages provide a general mechanism for checking for canceled messages. However, to request explicit information about the cancellation status of a particular message that was read from $RECEIVE, your server program can call the MESSAGESTATUS procedure. The MESSAGESTATUS procedure is the only way to test for cancellation if $RECEIVE is opened with a receive depth of 1.

The MESSAGESTATUS procedure returns a value that indicates whether a specified message has been canceled. You pass the message tag to the MESSAGESTATUS procedure to identify the message you are inquiring about.

The following example checks the status of the message whose message tag is MSG^TAG:

```
STATUS := MESSAGESTATUS(MSG^TAG);
```

The value returned in STATUS is one of the following:

1	The specified message has been canceled. Thus, a REPLY[X] call is still required but processing of the request should be terminated. There is no need to supply data with the reply.
0	The specified message has not been canceled. Thus, the server must process the message and send the REPLY[X] to the requester process.
-1	The specified message does not exist. No reply call is required.

Receiving and Processing System Messages

Recall that in addition to receiving messages from other processes, a process may receive system messages from the operating system on the $RECEIVE file.

Of the many system messages that the operating system can send, the writer of an application usually need be aware of only a subset. Of the system messages that a process typically processes, some are implicit and others are explicit. Implicit system messages indicate that some condition has occurred that may affect this process, such as the death of a process that was created by this process.

Explicit system messages result from an operation performed by another process on the process file, such as opening the file or performing a SETMODE function. This subsection discusses explicit messages.

The type of a system message is indicated in the first word of the message. Explicit messages include the following:

-32 (Control message)	Another process issued a CONTROL procedure call against this process.
-33 (Setmode message)	Another process issued a SETMODE procedure call against this process.
-34 (Resetsync message)	Another process issued a RESETSYNC procedure call against this process.
-35 (Controlbuf message)	Another process issued a CONTROLBUF procedure call against this process.
-37 (Setparam message)	Another process issued a SETPARAM procedure call against this process.
-103 (Open message)	Another process attempted to open this process.
-104 (Close message)	Another process attempted to close this process.

The Open and Close messages can be useful for several reasons; for example:

- To monitor how many processes have this process open
- To limit which processes are allowed to open the server
- To properly service a requester that is run as a fault-tolerant process pair

The Control, Setmode, Setparam, and Controlbuf messages are used if your program simulates an I/O device. Simple examples of handling these messages are given here. For a detailed example of handling the Open and Close system messages, see Section 22, Writing a Server Program. For details of I/O device simulation, see Section 24, Writing a Terminal Simulator.

The *Guardian Procedure Errors and Messages Manual* provides the format and recommended response for every system message that the operating system might generate.

Receiving System Messages

To receive system messages, your program needs to perform the following operations:

- Open $RECEIVE so that your program is able to receive system messages.
- Choose to read D-series system messages (the default) or C-series system messages.
- Read system messages.

The following paragraphs describe how to perform these operations.

Opening $RECEIVE to Receive System Messages

You can choose to receive or not to receive system messages in the $RECEIVE file. The choice is made when you open $RECEIVE with the FILE_OPEN_ procedure. If bit 15 of the *`options`* parameter is equal to 0 (the default value), then your server process will receive system messages:

```
FILE^NAME ':=' "$RECEIVE";
LENGTH := 8;
OPTIONS := 0;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    RECV^NUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    !sync^depth!,
                    OPTIONS);
IF ERROR <> 0 THEN ...
```

If bit 15 of the *`options`* parameter is set to 1, then your process will not receive system messages:

```
FILE^NAME ':=' "$RECEIVE";
LENGTH := 8;
OPTIONS.<15> := 1;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    RECV^NUM,
                    !access!,
                    !exclusion!,
                    !nowait^depth!,
                    !sync^depth!,
                    OPTIONS);
IF ERROR <> 0 THEN ...
```

Whether you choose to receive system messages affects the time at which the corresponding open in the requester finishes:

- If the server opens $RECEIVE without requesting system messages, the requester's open finishes as soon as the server has opened $RECEIVE.
- If the server opens $RECEIVE to request system messages and to enable two-way communication (*`receive-depth`* parameter set to a value greater than zero), the requester's open finishes when the server replies to the Open message.
- If the server opens $RECEIVE to request system messages but does not enable two-way communication (*`receive-depth`* parameter set to zero), the requester's open finishes when the server reads the Open message.

Receiving C-Series or D-Series System Messages

The *`options`* parameter also determines whether your process will read D-series system messages or C-series system messages. For example, message -103 is the D-series Open message, and message -30 is the equivalent C-series Open message. By default, you receive only the D-series messages. However, you can receive the C-series messages instead by setting bit 14 of the *`options`* parameter to 1.

The remainder of this section assumes D-series messages.

Reading System Messages

When you receive a system message from $RECEIVE, the READUPDATE[X] or READ[X] procedure returns a warning condition (CCG). Currently, a system message is the only reason why the READUPDATE[X] or READ[X] procedure returns with CCG. However, we recommend testing for error number 6 (system message received) in case other reasons for returning CCG are added in the future.

```
CALL READUPDATEX(RECV^NUM,
                 SBUFFER,
                 RCOUNT);
IF <> THEN
BEGIN
   CALL FILE_GETINFO_(RECV^NUM,
                      ERROR);
   IF ERROR = 6 THEN
   BEGIN
      !Process the system message
       .
       .
   END;
END;
```

The first word in the buffer returned by the read operation contains the system message number. Your program should then respond according to the system message number.

Processing Open and Close System Messages

Message number -103 (the Open message) is delivered to a process when another process tries to open the process (using the FILE_OPEN_, OPEN, or OPEN^FILE procedure). Similarly, message number -104 (the Close message) is delivered to a process when another process tries to close the process (either explicitly using FILE_CLOSE_, CLOSE, or CLOSE^FILE or implicitly by calling PROCESS_STOP_, STOP, or ABEND).

You might want your program to receive the Open and Close system messages if your program is a server to more than one requester process. This way, your program can control the number of requester processes that simultaneously have the server process open.

The following example allows up to 5 processes to have the server process open at one time:

```
LITERAL LIMIT^REACHED = 5;

ERROR^CODE := 0;
CASE BUFFER OF
BEGIN

   !Process the Open system message:
   ZSYS^VAL^SMSG^OPEN -> BEGIN
      IF OPENERS >= LIMIT^REACHED
      THEN ERROR^CODE := 12
      ELSE OPENERS := OPENERS + 1;
   END;

   !Process the Process Close system message:
   ZSYS^VAL^SMSG^CLOSE -> BEGIN
      OPENERS := OPENERS - 1;
   END;

   !Reject any other system message:
   OTHERWISE BEGIN                    !returns with
      ERROR^CODE := 2;                ! ERROR^CODE = 2
   END;
END;

!Reply to the sender:
CALL REPLYX(!buffer!,
            !write^count!,
            !count^written!,
            !message^tag!,
            ERROR^CODE);
```

This example uses the variable OPENERS to indicate how many processes currently have this process open. When the process receives an Open message, it adds one to OPENERS. When the process receives a Close message it subtracts one from OPENERS. Once the limit of five has been reached, then the process rejects the open with error number 12.

Processing Control, Setmode, Setparam, and Controlbuf Messages

Your process should accept the -32 (Control), -33 (Setmode), -37 (Setparam), -34 (Resetsync), and -35 (Controlbuf) messages only if the process is simulating an I/O device. In other words, some other process will issue CONTROL, SETMODE, SETPARAM, or CONTROLBUF procedure calls against this process as if the process were an I/O device.

For information on how to respond to each of these system messages, refer to the *Guardian Procedure Errors and Messages Manual*. The following example provides a skeletal outline:

```
CASE BUFFER OF
BEGIN
   -32 -> BEGIN
      !Process Control message;
      !For application-defined protocols,
      !   set REPLY^LEN as appropriate.
   END;

   -33 -> BEGIN
      !Process Setmode message
      !If last-params requested,
      !   set REPLY^LEN as appropriate.
   END;

   -35 -> BEGIN
      !Process Controlbuf message;
      !For application-defined protocols,
      !   set REPLY^LEN as appropriate.
   END;

   -37 -> BEGIN
      !Process Setparam message;
      !If last-params requested,
      !   set REPLY^LEN as appropriate.
   END;

   OTHERWISE -> BEGIN
      !Process any other message;
      !set REPLY^LEN as approproate.
   END;
END;

!Reply to the message:
CALL REPLYX(SBUFFER,
            $MIN(REPLY^LEN, RECEIVE^INFO[1]),
            !count^written!,
            !message^tag!,
            ERROR^CODE);
```

Handling Errors

For the $RECEIVE file, there are no error conditions for which error recovery should be attempted, except error 40 (operation timed out).

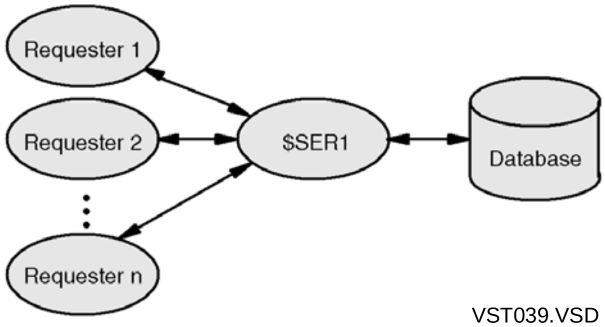
For a process file opened with a sync depth greater than zero, there are no error conditions for which error recovery should be retried, except error 40.

For a process file opened with a sync depth of zero, an operation that returns error 201 (path down) should be retried once if the process file is a process pair. An occurrence of error 201 means that the primary process failed. A reexecution of the call that returned the error causes communication to occur with the backup process, if any. If no backup process exists, a second error 201 is returned on reexecution of the call. At this point, the error can be considered fatal.

Communicating With Processes: Sample Programs

The sample programs shown here perform the same functions as the key-sequenced file programming example shown in Section 5, Communicating With Disk Files. Here, the program has been split into two programs: a requester program that handles input from and output to a terminal, and a server program that controls access to the database. The programs communicate through the $RECEIVE file. Figure 6-7 shows the relationship.

Figure 6-7. Example of a Requester/Server Application



To run the application, you need to create the database file and start the server and requester processes.

Create the database file using FUP commands as follows:

```
1> FUP
-SET TYPE K
-SET BLOCK 2048
-SET REC 135
-SET IBLOCK 2048
-SET KEYLEN 6
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES)
    REC 130
    BLOCK 2048
    IBLOCK 2048
    KEYLEN 6
    KEYOFF 0
    MAXEXTENTS 16
-CREATE $APPL.SUBAPPL.RECFILE
CREATED - $APPL.SUBAPPL.RECFILE
-EXIT
2>
```

Because the requester process opens the server by the name $SER1, you need to run the server process by this name. The following TACL command does this:

```
1>  RUN server-object-file-name /NAME $SER1, NOWAIT/
```

You can now run as many requester processes as you like as follows:

```
4>  RUN requester-object-file-name
     .
     .
```

Programming the Requester

The requester program prompts the user for a function to perform. The user can choose from the following functions:

- Read a record
- Add a record
- Update an existing record
- Exit the program

The requester formulates a request from information entered by the user and sends a message to the server process containing the appropriate information: a function code and, for operations that imply writing to the database, the contents of a database record.

The requester receives a response from the server. The response depends on the function. For operations that imply reading the database, the response includes database records. For write operations, a response indicating that the write finished successfully is enough.

The MAIN procedure responds to the user’s selection by calling the appropriate procedure:

- The READ^RECORD procedure allows the user to read one record followed optionally by subsequent sequential reads. It prompts the user for a part number and then sends the part number, along with a function code for an approximate read, to the server process. The response from the server contains the first record with a key equal to or greater than the supplied part number or an indication that the file contains no such record.

  If the reply contains a record, then the READ^RECORD procedure calls the DISPLAY^RECORD procedure to display the record on the user’s terminal. If the reply contained an end-of-file indication, then the READ^RECORD procedure prints a “no such record” message on the user’s terminal.

  If the user chooses to read more records, then the procedure sends another read request to the server process using the key value returned by the last read operation. This time the function code is set for a read-next operation instead of an approximate read.

- The UPDATE^RECORD procedure displays the record for update before prompting the user for the updated information. First it prompts the user for the key to the record to be updated (the part number). It then sends the part number to the server process along with a read-exact function code. The response from the server is either the record that the user wants to update or an indication that the record does not exist.

  If the record does not exist, the procedure prints a diagnostic and returns control to the main procedure. If a record is returned, the procedure calls DISPLAY^RECORD to display the record on the user’s terminal then calls ENTER^RECORD to prompt the user to enter new values. Once the values are entered, UPDATE^RECORD sends the updated record to the server along with a write function code. The response from the server indicates that the write was successful, and control returns to the MAIN procedure.

- The INSERT^RECORD procedure calls the ENTER^RECORD procedure to prompt the user to enter a new record. The INSERT^RECORD procedure then sends this data structure to the server process along with a function code indicating that the server should write a new record. The response from the server is either a confirmation that the write was completed as intended or an indication that the write could not proceed because a record with the same key already exists. In either case, control is returned to the MAIN procedure. If the write could not be completed, INSERT^RECORD prints a message on the user’s terminal.

- The EXIT^PROGRAM procedure stops the requester program. The server continues to run because other requesters may still be using it.

The TAL code for the requester program follows.

```
?INSPECT, SYMBOLS, NOCODE
?NOLIST,SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN    = ZSYS^VAL^LEN^FILENAME;  !Maximum
                                             ! file-name
                                             ! length
LITERAL OLD        = 0;          !updating in ENTER^REC
LITERAL NEW        = 1;          !new record in ENTER^REC
LITERAL BUFSIZE    = 132;        !size of terminal buffer
LITERAL PARTSIZE   = 6;          !size of part number
LITERAL DESCSIZE   = 60;         !size of part description
LITERAL SUPPSIZE   = 60;         !size of supplier name
LITERAL READ^APPROX= 1;          !requester function
LITERAL READ^EXACT = 2;
LITERAL WRITE^ONE  = 3;
LITERAL UPDATE^ONE = 4;
LITERAL READ^NEXT  = 5;

STRING .SBUFFER[0:BUFSIZE];      !I/O buffer (one extra char)
STRING .S^PTR;                   !pointer to end of string
INT     SERVER^NUM;              !server file number
INT     TERMNUM;                 !terminal file number

!Data structure for receiving part records from the server
!process:

STRUCT .PART^RECORD;
BEGIN
   STRING PART^NUMBER[0:PARTSIZE-1];
   STRING DESCRIPTION[0:DESCSIZE-1];
   INT    DESC^LEN;
   STRING SUPPLIER[0:SUPPSIZE-1];
   INT    SUP^LEN;
   INT    ON^HAND;
   INT    UNIT^PRICE;
END;

!Data structure for sending a request to the server:

STRUCT .REQUEST;
BEGIN
   INT    REQUEST^FUNCTION;
   STRUCT PART(PART^RECORD);
   STRUCT OLD^PART(PART^RECORD);
END;
```

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,NUMIN,
?  PROCESS_STOP_,READX,DNUMOUT,FILE_GETINFO_,DNUMIN)
?LIST
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to format
! and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

   DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, when there is no file number for it.
! FILE^ERRORS is used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING .FNAME;
INT     LEN;
INT     ERROR;
BEGIN

!  Compose and print the message

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_;
END;


!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!-------------------------------------------------------------

PROC FILE^ERRORS (FNUM);
INT         FNUM;
BEGIN
   INT      ERROR;
   STRING  .FNAME[0:MAXFLEN - 1];
   INT      FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING  .BUF;
INT      LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
END;



!------------------------------------------------------------
! This procedure asks the user for the next function to do:
!
! "r" to read records
! "u" to update a record
! "i" to insert a record
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' to Read Record, ");
   PRINT^STR("     'u' to Update a Record,  ");
   PRINT^STR("     'i' to Insert a Record, ");
   PRINT^STR("     'x' to Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER, @S^PTR '-' @SBUFFER,
                  BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to display a part record on the terminal
!------------------------------------------------------------

PROC DISPLAY^RECORD;
BEGIN

   PRINT^BLANK;

!  Display part number:

   PRINT^STR("Part Number Is:    "
            & PART^RECORD.PART^NUMBER
              FOR PARTSIZE);

!  Display part description:

   PRINT^STR("Part Description:  "
            & PART^RECORD.DESCRIPTION
              FOR PART^RECORD.DESC^LEN);

!  Display part supplier name:

   PRINT^STR("Supplier:          "
            & PART^RECORD.SUPPLIER
              FOR PART^RECORD.SUP^LEN);

!  Display quantity on hand:

   START^LINE;
   PUT^STR("Quantity on hand:  ");
   PUT^INT(PART^RECORD.ON^HAND);
   PRINT^LINE;

!  Display unit price:

   START^LINE;
   PUT^STR("Unit Price:        $");
   PUT^INT(PART^RECORD.UNIT^PRICE);
   PRINT^LINE;
END;
```

```
!------------------------------------------------------------
! Procedure to prompt user for input to build a new record or
! update an existing record.  When updating, an empty
! response (COUNT^READ=0) means to leave the existing value
! unchanged.
!------------------------------------------------------------

PROC ENTER^RECORD(TYPE);
INT  TYPE;

BEGIN
   INT     COUNT^READ;
   INT     STATUS;
   STRING .NEXT^ADDR;

   DEFINE BLANK^FILL(F) =
            F ':=' " " & F FOR $LEN(F)*$OCCURS(F)-1 BYTES #;

   PRINT^BLANK;

!  If inserting a new record, prompt for a part number.
!  If updating an existing record, record number is already
!  known:

   IF TYPE = NEW THEN
   BEGIN
      SBUFFER ':=' "Enter Part Number:  " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      BLANK^FILL(REQUEST.PART.PART^NUMBER);
      REQUEST.PART.PART^NUMBER ':='
                     SBUFFER FOR $MIN(COUNT^READ,PARTSIZE);
   END;

!  If updating a record, copy the part number from the
!  record just read:

   IF TYPE = OLD THEN
      REQUEST.PART.PART^NUMBER ':=' PART^RECORD.PART^NUMBER
                                    FOR PARTSIZE;
```

```
!  Prompt for a part description:

   SBUFFER ':=' "Enter Part Description:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,DESCSIZE);
      BLANK^FILL(REQUEST.PART.DESCRIPTION);
      REQUEST.PART.DESCRIPTION ':=' SBUFFER
                                    FOR COUNT^READ;
      REQUEST.PART.DESC^LEN := COUNT^READ;
   END;

!  Prompt for the name of the supplier:

   SBUFFER ':=' "Enter Supplier Name:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,SUPPSIZE);
      BLANK^FILL(REQUEST.PART.SUPPLIER);
      REQUEST.PART.SUPPLIER ':=' SBUFFER
                                      FOR COUNT^READ;
      REQUEST.PART.SUP^LEN := COUNT^READ;
   END;

!  Prompt for the quantity on hand:

PROMPT^AGAIN:
   SBUFFER ':=' "Enter Quantity On Hand:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR := NUMIN(SBUFFER,REQUEST.PART.ON^HAND,10,
                          STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN;
      END;
   END;
```

```
!  Prompt for the unit price:

PROMPT^AGAIN1:
   SBUFFER ':=' "Enter Unit Price:  $" -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR := NUMIN(SBUFFER,REQUEST.PART.UNIT^PRICE,10,
                          STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN1;
      END;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r." The start of the read is selected by approximate key
! positioning.  The user has the option of sequentially
! reading subsequent records.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT         COUNT^READ;
   INT         ERROR;

!  Prompt the user for the part number:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

!  Fill in REQUEST^FUNCTION and part number parts of data
!  structure:

   REQUEST.REQUEST^FUNCTION := READ^APPROX;
   REQUEST.PART.PART^NUMBER ':=' [PARTSIZE*[" "]];
   REQUEST.PART.PART^NUMBER ':=' SBUFFER FOR COUNT^READ;

!  Request one record from the server.
!  If server replies with end-of-file indication,
!  return control to the main procedure.

   CALL WRITEREADX(SERVER^NUM,REQUEST,$LEN(REQUEST),
                   $LEN(PART^RECORD),COUNT^READ);

   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERVER^NUM,ERROR);
      IF ERROR = 1 THEN
      BEGIN
         PRINT^STR("No such record");
         RETURN;
      END;
      CALL FILE^ERRORS(SERVER^NUM);
   END;

!  Display record on terminal:

   PART^RECORD ':=' REQUEST FOR $LEN(PART^RECORD);
   CALL DISPLAY^RECORD;
```

```
!  Prompt user to read more records.  Return to MAIN
!  procedure unless the user types "y" or "Y":

   PRINT^BLANK;
   SBUFFER ':=' "Do you want to read another record (y/n)? "
                 -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                    BUFSIZE,COUNT^READ);
   IF NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y") THEN RETURN;

!  Loop reading and displaying records until user declines
!  to read the next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

   !  Set REQUEST^FUNCTION to get the next record:

      REQUEST.REQUEST^FUNCTION := READ^NEXT;

   !  Set PART.PART^NUMBER to the part number just read:

      REQUEST.PART.PART^NUMBER ':=' PART^RECORD.PART^NUMBER
                                         FOR PARTSIZE;

   !  Send request to server.
   !  If server replies with end-of-file indication,
   !  return control to the main procedure.

      CALL WRITEREADX(SERVER^NUM,REQUEST,$LEN(REQUEST),
                       $LEN(PART^RECORD));
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(SERVER^NUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(SERVER^NUM);
      END;

   !  Display the record on the terminal:

      PART^RECORD ':=' REQUEST FOR $LEN(PART^RECORD);
      CALL DISPLAY^RECORD;

      PRINT^BLANK;
      !  Prompt the user to read the next record (user
      !  must respond "y" to accept, otherwise return
      !  to select next function):

         SBUFFER ':=' ["Do you want to read another ",
                                     "record (y/n)? "]
                        -> @S^PTR;
         CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                         BUFSIZE,COUNT^READ);
         IF <> THEN CALL FILE^ERRORS(TERMNUM);

         SBUFFER[COUNT^READ] := 0;
      END
      UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
   END;



   !-------------------------------------------------------------
   ! Procedure for updating a record.  The user selected
   ! function "u."  The user is prompted to enter the part
   ! number of the record to be updated, then the old contents
   ! are displayed on the user's terminal before the user is
   ! prompted to enter the updated record.
   !-------------------------------------------------------------

   PROC UPDATE^RECORD;
   BEGIN

      INT     COUNT^READ;
      INT     ERROR;
      STRUCT .SAVE^REC(PART^RECORD);
      STRUCT .CHECK^REC(PART^RECORD);

      PRINT^BLANK;

   !  Prompt the user for the part number of the record to be
   !  updated:

      PRINT^BLANK;
      SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   !  Fill in the request to read the part record:

      REQUEST.PART.PART^NUMBER ':=' [PARTSIZE*[" "]];
      REQUEST.PART.PART^NUMBER ':=' SBUFFER FOR COUNT^READ;
      REQUEST.REQUEST^FUNCTION := READ^EXACT;
```

```
!  Send the request to the server.  If no such record exists,
!  the procedure informs the user and returns control to
!  the main procedure:

   CALL WRITEREADX(SERVER^NUM,REQUEST,$LEN(REQUEST),
                   $LEN(PART^RECORD),COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERVER^NUM,ERROR);
      IF ERROR = 11 OR ERROR = 1 THEN
      BEGIN
         PRINT^BLANK;
         START^LINE;
         PUT^STR("No such record");
         PRINT^LINE;
         RETURN;
      END

      ELSE CALL FILE^ERRORS(SERVER^NUM);
   END;

!  Save the record in the REQUEST structure for later
!  comparison by the server:

   PART^RECORD ':=' REQUEST FOR $LEN(PART^RECORD) BYTES;
   REQUEST.OLD^PART ':=' PART^RECORD FOR $LEN(PART^RECORD)
                         BYTES;

   REQUEST.PART ':=' PART^RECORD.PART^NUMBER FOR PARTSIZE;

!  Display the record on the terminal:

   CALL DISPLAY^RECORD;

!  Prompt the user for the updated record:

   CALL ENTER^RECORD(OLD);

!  Fill in the part number:

   REQUEST.PART.PART^NUMBER ':=' PART^RECORD.PART^NUMBER
                                 FOR PARTSIZE;
```

```
!  Now that we have the user's changes, send a request to
!  the server to have the file updated.  The server uses
!  the REQUEST.OLD information to determine if the
!  record has been updated while the user was responding:

   REQUEST.REQUEST^FUNCTION := UPDATE^ONE;

   CALL WRITEREADX(SERVER^NUM,REQUEST,$LEN(REQUEST),
                   $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERVER^NUM,ERROR);
      IF ERROR = 300 THEN
      BEGIN
         PRINT^STR("The record was changed by someone " &
                   "else while you were working on it.");
         PRINT^STR("Your change was not made.");
         RETURN;
      END
      ELSE CALL FILE^ERRORS(SERVER^NUM);
   END;
   PRINT^STR("Your changes have been made ");
END;
```

```
!------------------------------------------------------------
! Procedure for inserting a record.  The user selected
! function "i." The user is prompted to enter the new record.
! The procedure inserts the new record in the appropriate
! place in the file.
!------------------------------------------------------------

PROC INSERT^RECORD;
BEGIN
   INT     ERROR;

   PRINT^BLANK;

!  Set the REQUEST^FUNCTION:

   REQUEST.REQUEST^FUNCTION := WRITE^ONE;

!  Prompt the user for the new record:

   CALL ENTER^RECORD(NEW);

!  Send the new record to the server:

   CALL WRITEREADX(SERVER^NUM,REQUEST,$LEN(REQUEST),
                   $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERVER^NUM,ERROR);
      IF ERROR = 10 THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR
            ("Already a record with that part number.");
         PRINT^STR("Your new one was not entered.");
      END
      ELSE BEGIN
         CALL FILE^ERRORS(SERVER^NUM);
      END;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure to exit the program.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "u", "a," or "x."
!------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that his selection was invalid
!  then return to prompt again for a valid function:

   PRINT^STR("INVALID COMMAND: "  &
             "Type either 'r,' 'u,' 'i,' or 'x'");
END;
```

```
!-------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .SERVER^NAME[0:MAXFLEN - 1];   !name of server
                                         ! process
   INT     SERVERLEN;                    !length of server
                                         ! name
   STRING .TERMNAME[0:MAXFLEN - 1];      !terminal file
   INT     TERMLEN;                      !length of terminal-
                                         ! file name
   INT     ERROR;

!  Read and discard startup messages.

   CALL INITIALIZER;

!  Open the terminal file. For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  from the Startup message; see Section 8 for details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the server process:

   SERVER^NAME ':=' "$SER1" -> @S^PTR;
   SERVERLEN := @S^PTR '-' @SERVER^NAME;
   ERROR := FILE_OPEN_(SERVER^NAME:SERVERLEN,
                       SERVER^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(SERVER^NAME:
                            @S^PTR '-' @SERVER^NAME,
                            ERROR);
END;
```

```
!--------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize, then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!--------------------------------------------------------------

PROC PARTS MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function x:

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command.

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "u", "U"  -> CALL UPDATE^RECORD;

         "i", "I"  -> CALL INSERT^RECORD;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^COMMAND;
      END;
   END;
END;
```

Programming the Server

The server program reads and processes messages that arrive in the $RECEIVE file. Each message contains a function code that the server process uses to determine what action to take. If the action involves writing a record to the database, then the message also contains the database record to be written. The possible functions are:

- Read-approximate record
- Read the next record
- Read the exact record
- Update an existing record
- Write a new record

The MAIN procedure of the server process calls the INIT procedure and then reads the incoming message. It then calls another procedure depending on the value of the received function code.

- The INIT procedure disposes of the startup messages. It also opens the home terminal and the data file for the program. The data file is identified by a CLASS MAP DEFINE. You must create the CLASS MAP DEFINE as follows:

  ```
  1> SET DEFINE CLASS MAP, FILE $APPL.SUBAPPL.RECFILE
  2> ADD DEFINE =PARTFILE
  ```

- The READ^APPROX^RECORD procedure is called when a requester process makes its first of a sequence of read requests. This procedure uses the part number supplied in the message as the primary key. If no such key exists, then the procedure reads the record with the next higher key. The procedure then returns this record to the message sender.

- The READ^NEXT^RECORD procedure is called when the requester process asks to read the next record in a sequence that started with an approximate read. This procedure uses the part number supplied in the message as the primary key and reads the record with the next higher key. The server then returns this record to the message sender.

- The READ^EXACT^RECORD procedure is called when a requester process wants to update a database record. It uses the part number supplied in the message as the primary key and returns the corresponding record to the requester process. If no such key exists, the server returns an error condition instead of the record.

- The UPDATE^RECORD procedure is called when a requester wants to update a database record. The procedure uses the part number supplied in the message as the primary key to the database file, then it overwrites the corresponding database record using the database record also supplied in the message.

  Note that to update a record, the requester invokes the server twice: once to read the record and once to update it. When making the update request, the requester sends the read record back to the server so that UPDATE^RECORD can compare it with the current record value. It does this to ensure that the record was not changed while the user was entering the new data.

- The WRITE^RECORD procedure is called when the requester process sets the function code to insert a new record. The procedure extracts the database record from the incoming message and writes it to the database file. If a record with the same key already exists, then the procedure returns an error indication to the requester and the new record is discarded.

The TAL code for the server program appears on the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST,SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;  !Maximum file-name
                                          ! length
LITERAL OLD     = 0;             !updating in ENTER^REC
LITERAL NEW     = 1;             !new record in ENTER^REC
LITERAL BUFSIZE = 132;           !size of terminal buffer
LITERAL PARTSIZE= 6;             !size of part number
LITERAL DESCSIZE= 60;            !size of part description
LITERAL SUPPSIZE= 60;            !size of supplier name
LITERAL READ^APPROX= 1;          !function values:
LITERAL READ^EXACT = 2;
LITERAL WRITE^ONE  = 3;
LITERAL UPDATE^ONE = 4;
LITERAL READ^NEXT  = 5;

STRING .SBUFFER[0:BUFSIZE];      !I/O buffer (one extra char)
STRING .S^PTR;                   !pointer to end of string
INT     PARTFILE^NUM;            !part file number
INT     TERMNUM;                 !terminal file number
INT     RECV^NUM;                !$RECEIVE file number
INT     REPLY^ERROR;             !error returned to requester

!Structure for part records:

STRUCT .PART^RECORD;
BEGIN
   STRING PART^NUMBER[0:PARTSIZE-1];
   STRING DESCRIPTION[0:DESCSIZE-1];
   INT    DESC^LEN;
   STRING SUPPLIER[0:SUPPSIZE-1];
   INT    SUP^LEN;
   INT    ON^HAND;
   INT    UNIT^PRICE;
END;

!Structure for messages received from requester:

STRUCT .REQUEST;
BEGIN
   INT    REQUEST^FUNCTION;
   STRUCT PART(PART^RECORD);
   STRUCT OLD^PART(PART^RECORD);
END;
```

```
?NOLIST, SOURCE  $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,REPLYX,
?  KEYPOSITION,PROCESS_STOP_,READX,FILE_GETINFO_,
?  READUPDATEX,READUPDATELOCKX,WRITEUPDATEUNLOCKX,
?  FILE_GETINFOLIST_,UNLOCKREC,DNUMOUT)
?LIST


!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to format
! and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

   DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, when there is no file number for it.
! FILE^ERRORS is used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING .FNAME;
INT     LEN;
INT     ERROR;
BEGIN

!  Compose and print the message

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS (FNUM);
INT         FNUM;
BEGIN
   INT      ERROR;
   STRING  .FNAME[0:MAXFLEN - 1];
   INT      FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING  .BUF;
INT      LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! This procedure reads one record from the data file.
! It is invoked in response to a READ^APPROX request issued
! by a requester process.
!------------------------------------------------------------

PROC READ^APPROX^RECORD;
BEGIN
   INT         COUNT^READ;
   INT         ERROR;

!  Position approximately to the selected record:

   CALL KEYPOSITION(PARTFILE^NUM,
                    REQUEST.PART.PART^NUMBER,
                    !key^specifier!,
                    !length^word!,
                    0);
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record:

   CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,REPLY^ERROR);
      RETURN;
   END;
   REPLY^ERROR := 0;
END;
```

```
!------------------------------------------------------------
! This procedure reads one record from the data file.
! It is invoked in response to a READ^EXACT request issued by
! a requester process, as the first phase of a record update.
!------------------------------------------------------------

PROC READ^EXACT^RECORD;
BEGIN
   INT         COUNT^READ;
   INT         ERROR;

!  Position exactly to the selected record:

   CALL KEYPOSITION(PARTFILE^NUM,
                    REQUEST.PART.PART^NUMBER,
                    !key^specifier!,
                    !length^word!,
                    2);
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record:

   CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,REPLY^ERROR);
      RETURN;
   END;
   REPLY^ERROR := 0;
END;


!------------------------------------------------------------
! Procedure to insert a new record into the data file.
!------------------------------------------------------------

PROC WRITE^RECORD;
BEGIN

   CALL WRITEX(PARTFILE^NUM,REQUEST.PART,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,REPLY^ERROR);
      RETURN;
   END;
   REPLY^ERROR := 0;
END;
```

```
!------------------------------------------------------------
! Procedure for updating a record.  The procedure first
! reads the record from the data file and checks it against
! the original value received by the requester to make sure
! it has not been updated by another user.
! The procedure then writes the updated record to the file.
!------------------------------------------------------------

PROC UPDATE^RECORD;
BEGIN

   INT     COUNT^READ;
   INT     ERROR;
   STRUCT .SAVE^REC(PART^RECORD);
   STRUCT .CHECK^REC(PART^RECORD);

!  Position exactly to the selected record.

   CALL KEYPOSITION(PARTFILE^NUM,
                    REQUEST.PART.PART^NUMBER,
                    !key^specifier!,
                    !length^word!,
                    2);
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record.  If no such record exists,
!  the procedure sets the reply error to the file system
!  error number and returns to the main procedure:

   CALL READUPDATELOCKX(PARTFILE^NUM,PART^RECORD,
                        $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,REPLY^ERROR);
      RETURN;
   END;

!  Check that the record just read is identical to the record
!  read earlier in the update record sequence.  If not,
!  return with REPLY^ERROR set to 300:

   IF PART^RECORD <> REQUEST.OLD^PART FOR
                     $LEN(PART^RECORD) BYTES THEN
   BEGIN
      CALL UNLOCKREC(PARTFILE^NUM);
      REPLY^ERROR := 300;
      RETURN;
   END;
```

```
!  Write the new record to the file:

   CALL WRITEUPDATEUNLOCKX(PARTFILE^NUM,REQUEST.PART,
                           $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,REPLY^ERROR);
      RETURN;
   END;
   REPLY^ERROR := 0;
END;


!------------------------------------------------------------
! Procedure to read the next record from the data file.
! The requester supplied the part number of the last
! record read.
!------------------------------------------------------------

PROC READ^NEXT^RECORD;
BEGIN
   INT     ERROR;
   INT     COUNT^READ;
   INT     POSITIONING^MODE;

!  Position approximately to the selected record, unless it
!  is the exact record:

   POSITIONING^MODE := %B1000000000000000;
   CALL KEYPOSITION(PARTFILE^NUM,REQUEST.PART.PART^NUMBER,
                    !key^specifier!,
                    !length^word!,
                    POSITIONING^MODE);
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record:

   CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM, REPLY^ERROR);
      RETURN;
   END;
   REPLY^ERROR := 0;
END;
```

```
!-------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .PARTFILE^NAME[0:MAXFLEN - 1]; !name of part file
   INT     PARTFILE^LEN;                 !length of part-file
                                         ! name
   STRING .TERMNAME[0:MAXFLEN - 1];      !terminal file
   INT     TERMLEN;                      !length of terminal-
                                         ! file name
   STRING .RECV^NAME[0:MAXFLEN - 1];     !$RECEIVE file name
   INT     RECVLEN;                      !length of string
   INT     OPTIONS;                      !for FILE_OPEN_
                                         ! procedure
   INT     RECV^DEPTH;                   !for $RECEIVE
   INT     SYNCH^DEPTH;                  !for data file
   INT     ERROR;

!  Read and discard startup messages.

   CALL INITIALIZER;

!  Open the terminal file. For simplicity we use the home
!  terminal; the recommended approach is to use the IN file
!  read from the Startup message; see Section 8 for
!  details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open $RECEIVE with a receive depth of 1 and to reject
!  system messages:

   RECV^NAME ':=' "$RECEIVE" -> @S^PTR;
   RECVLEN := @S^PTR '-' @RECV^NAME;
   OPTIONS := %B0000000000000001;
   RECV^DEPTH := 1;
   ERROR := FILE_OPEN_(RECV^NAME:RECVLEN,RECV^NUM,
                      !access!,
                      !exclusion!,
                      !nowait^depth!,
                      RECV^DEPTH,OPTIONS);
   IF ERROR <> 0 THEN
            CALL FILE^ERRORS^NAME(RECV^NAME:RECV^NUM,ERROR);

!  Open the part file with a sync depth of 1:

   PARTFILE^NAME ':=' "=PARTFILE" -> @S^PTR;
   PARTFILE^LEN := @S^PTR '-' @PARTFILE^NAME;
   SYNCH^DEPTH := 1;
   ERROR := FILE_OPEN_(PARTFILE^NAME:PARTFILE^LEN,
                      PARTFILE^NUM,
                      !access!,
                      !exclusion!,
                      !nowait^depth!,
                      SYNCH^DEPTH);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(PARTFILE^NAME:PARTFILE^LEN,
                           ERROR);
END;
```

```
!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize, then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!-------------------------------------------------------------

PROC PARTS MAIN;
BEGIN

   CALL INIT;

!  Loop indefinitely:

   WHILE 1 DO
   BEGIN

   !  Read a message from $RECEIVE:

      CALL READUPDATEX(RECV^NUM,REQUEST,$LEN(REQUEST));
      IF <> THEN CALL FILE^ERRORS(RECV^NUM);

   !  Select procedure based on request function:

      CASE REQUEST.REQUEST^FUNCTION OF
      BEGIN

         READ^APPROX  -> CALL READ^APPROX^RECORD;

         READ^EXACT   -> CALL READ^EXACT^RECORD;

         WRITE^ONE    -> CALL WRITE^RECORD;

         UPDATE^ONE   -> CALL UPDATE^RECORD;

         READ^NEXT    -> CALL READ^NEXT^RECORD;

         OTHERWISE    -> CALL PROCESS_STOP_;
      END;

   !  Send the reply back to the requester:

      CALL REPLYX(PART^RECORD,$LEN(PART^RECORD),
                  !count^written!,
                  !message^tag!,
                  REPLY^ERROR);
      IF <> THEN CALL PROCESS_STOP_;
   END;
END;
```

7 Using DEFINEs

DEFINEs are file-system elements that provide a means for passing information to a process. For example, you can use DEFINEs to pass attributes to a process to provide:

- An alternate name for accessing a file
- A list of subvolumes to search for a file name
- A simple way to set up attributes for labeled-tape processing
- A simple means of passing attributes to the spooler subsystem

You can make use of DEFINEs interactively using TACL, or you can work with DEFINEs programmatically. DEFINEs are stored within the program file segment (PFS) of the creating process. DEFINEs specified interactively are stored in the PFS of the TACL process and affect the environment of TACL. The programmatic approach stores DEFINEs in the context of the creating process and affects the environment of the creating process. In either case, DEFINEs can be passed on to other processes when creating new processes. This functionality is also supported in SCF for a generic process. For more information, see the *SCF Reference Manual for the Kernel Subsystem*.

This section discusses how to use DEFINEs programmatically. For details on how to use DEFINEs with TACL, read the *Guardian User's Guide*.

The subsection Working With DEFINEs describes what your process needs to do to make use of information passed to the process in the form of DEFINEs.

The subsection Adding DEFINEs describes how to add DEFINEs to the process context by setting up a **working set** of DEFINE attributes, how to name the set of attributes, and how to save the working set in the process context. Specifically, you will learn how to use procedure calls to perform the following tasks:

- Control whether your process can create or use DEFINEs (DEFINEMODE procedure).
- Set attributes in the working set (DEFINESETATTR and DEFINESETLIKE procedures).
- Check the working set for errors (DEFINEVALIDATEWORK procedure).
- Assign a name to a group of DEFINE attributes, thereby adding the DEFINE to the context of your process (DEFINEADD procedure).
- Delete DEFINEs from your process context (DEFINEDELETE and DEFINEDELETEALL procedures).
- Save and restore a working set of DEFINE attributes (DEFINESAVE[WORK[2]] and DEFINERESTORE[WORK[2]] procedures), and use these techniques for improving the performance of your program.

Section 16, Creating and Managing Processes, also contains additional information about DEFINEs. It describes how to use the PROCESS_LAUNCH_ procedure to control how DEFINEs are passed from one process to another during process creation.

For complete details of the syntax of all DEFINE-related procedure calls, see the *Guardian Procedure Calls Reference Manual*.

Example Uses for DEFINEs

DEFINEs allow attributes to be grouped and named. These attributes can then be passed to a process or retrieved simply by specifying the name of the DEFINE. DEFINEs remove the need to set up attributes each time a given process is invoked.

Specifically, several classes of DEFINEs each pass attributes to a specific process or class of processes. Examples of classes of DEFINEs include:

- CLASS MAP DEFINEs
- CLASS SEARCH DEFINEs
- CLASS TAPE DEFINEs
- CLASS DEFAULTS DEFINEs

CLASS MAP DEFINEs enable a file to be accessed by a DEFINE name as well as the file name. That is, the DEFINE name is mapped to the file name. The DEFINE name can then be passed to the file system instead of the file name. In other words, you can use a CLASS MAP DEFINE to propagate a file name from the process's creator without using the startup sequence of messages. This file name can then be passed to procedures such as FILE_OPEN_ in the form of a DEFINE name, removing the need to hard code the file name.

CLASS SEARCH DEFINEs contain information to be used for resolving file names with a search list. It has 21 attributes named SUBVOL0 through SUBVOL20 and another 21 attributes named RELSUBVOL0 through RELSUBVOL20. Each of these attributes takes the same form and is optional. The value of one attribute is either a single subvolume specification or a list of them enclosed in parentheses and separated by commas. A subvolume specification can be a fully or partially qualified subvolume name, or the name of a CLASS DEFAULTS DEFINE.

With the SUBVOL*nn* attributes, subvolume name resolution takes place when the attribute is added; with the RELSUBVOL*nn* attributes, subvolume name resolution takes place when the DEFINE is used. The search order for a CLASS SEARCH DEFINE is as follows:

SUBVOL0
RELSUBVOL0
SUBVOL1
RELSUBVOL1
. . .

SUBVOL20
RELSUBVOL20

The name of the DEFINE can then be used in a call to the FILENAME_RESOLVE_ procedure to provide a search list for a program file you want to execute, or it can be used to provide a search list for locating component source files when compiling a program.

If any attribute is a list, the search order is from left to right within the list.

CLASS TAPE DEFINEs contain attribute information for use with labeled magnetic tapes. One CLASS TAPE DEFINE must be used for each labeled-tape file that is accessed by your application. CLASS TAPE DEFINEs are processed by the tape process and by the file-system FILE_OPEN_ procedure. The attribute parameters of a CLASS TAPE DEFINE specify parameters such as block size and density.

CLASS DEFAULTS DEFINEs are used to pass default system, volume, subvolume, and swap information to a process. (For TNS/R native processes, the swap information is ignored.)

The following paragraphs provide examples of each of these classes of DEFINEs.

Example of a CLASS MAP DEFINE

The following example sets up a DEFINE named =MYFILE with the file name \SYS1.$OURVOL.MYSUBVOL.DIARY. The code uses procedures DEFINESETATTR and DEFINEADD. The file is later accessed using the DEFINE name:

```
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "MAP";
ATTRIBUTE^LENGTH := 3;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "FILE            ";
ATTRIBUTE^VALUE ':=' "\SYS1.$OURVOL.MYSUBVOL.DIARY";
ATTRIBUTE^LENGTH := 28;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

DEFINE^NAME ':=' "=MYFILE                  ";
LENGTH := 7;
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
 .
 .
 .
ERROR := FILE_OPEN_(DEFINE^NAME:LENGTH,
                    FILENUM);
IF ERROR <> 0 THEN ...
```

Example of a CLASS SEARCH DEFINE

The following example sets up attributes for a DEFINE named =_PROGRAM_SEARCH using procedures DEFINESETATTR and DEFINEADD. Later, the program calls the FILENAME_RESOLVE_ procedure to find the name within the search paths set up by the search DEFINE. FILENAME_RESOLVE_ returns the fully qualified file name.

```
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "SEARCH";
ATTRIBUTE^LENGTH := 6;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "SUBVOL0         ";
ATTRIBUTE^VALUE ':=' "=_DEFAULTS";
ATTRIBUTE^LENGTH := 10;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "SUBVOL1         ";
ATTRIBUTE^VALUE ':=' "$APPPROG.MYPROGS,$APPPROG.YOURPGS";
ATTRIBUTE^LENGTH := 33;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

DEFINE^NAME ':=' "=_PROGRAM_SEARCH        "
LENGTH := 16;
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
 .
 .
 .
CALL FILENAME_RESOLVE_(OBJFILE:P^LEN,
                       FULL^NAME:MAX^LEN,
                       FULL^LEN,
                       !options!,
                       !override^name:length!,
                       DEFINE^NAME:LENGTH);
```

Refer to Section 13, Manipulating File Names, for details of the FILENAME_RESOLVE_ procedure and other procedures that manipulate file names.

Example of a CLASS TAPE DEFINE

The following example sets up a DEFINE called =ANSITAPE1. When the DEFINE is later passed to the FILE_OPEN_ procedure, the corresponding tape file is opened with the DEFINE attributes automatically set.

```
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "TAPE";
ATTRIBUTE^LENGTH := 4;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "USE             ";
ATTRIBUTE^VALUE ':=' "IN";
ATTRIBUTE^LENGTH := 2;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "VOLUME          ";
ATTRIBUTE^VALUE ':=' "XT55";
ATTRIBUTE^LENGTH := 4;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "LABELS          ";
ATTRIBUTE^VALUE ':=' "ANSI";
ATTRIBUTE^LENGTH := 4;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

DEFINE^NAME ':=' "=ANSITAPE1              ";
LENGTH := 10;
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
 .
 .
ERROR := FILE_OPEN_(DEFINE^NAME:LENGTH,
                    FILENUM);
IF ERROR <> 0 THEN ...
```

CLASS DEFAULTS DEFINEs

CLASS DEFAULTS DEFINEs are used to pass default system, volume, subvolume, and swap information to a process. The receiving process uses the system, volume, and subvolume default values to expand any file names that are not fully qualified.

(Although swap space is handled by the Kernel-Managed Swap Facility (KMSF), a swap file name can still be passed in the =_DEFAULTS DEFINE. This feature supports programs that use the swap file name to determine the volume on which to create temporary files.)

The following example sets up the =_DEFAULTS DEFINE with the subvolume $OURVOL.ACCOUNTS. These default values are automatically passed on to child processes:

```
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "DEFAULTS";
ATTRIBUTE^LENGTH := 8;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

ATTRIBUTE^NAME ':=' "VOLUME          ";
ATTRIBUTE^VALUE ':=' "$OURVOL.ACCOUNTS";
ATTRIBUTE^LENGTH := 16;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...
```

=_DEFAULTS is a special case of DEFINE and should be handled differently from other DEFINEs. Specifically:

- Your application need not reference the =_DEFAULTS DEFINE. Many procedures use the default values specified in the =_DEFAULTS DEFINE; see the *Guardian Procedure Calls Reference Manual* to check which procedures use the defaults and which do not.
- The file system automatically modifies the =_DEFAULTS DEFINE of a process when the process receives its Startup message from the command interpreter. The file system assigns values from this Startup message to the VOLUME attribute of the DEFINE. This often has no effect because, for example, if TACL sends the Startup message, it will have ensured that the DEFINE and the Startup message contain the same default volume. However, programs other than TACL might behave differently. See Section 8, Communicating With a TACL Process, for details about the startup sequence.
- The =_DEFAULTS DEFINE cannot be deleted.
- The name of the =_DEFAULTS DEFINE cannot be changed.
- The =_DEFAULTS DEFINE is always propagated to other processes regardless of the setting of the DEFINE mode. See Section 16, Creating and Managing Processes, for a discussion on how DEFINEs are propagated.

DEFINE Names

You have seen DEFINE names like =MYFILE and =ANSITAPE1 in the previous subsection. You specify the name when creating the DEFINE, and you use the name to subsequently identify the DEFINE. (The above examples show DEFINE creation using the DEFINEADD procedure.)

A DEFINE name must conform to the following rules:

- The name must be 2 through 24 characters long.
- The first character must be an equal sign (=).
- The second character must be a letter. (DEFINE names whose second character is an underscore are reserved for uses defined by Tandem. The =_DEFAULTS DEFINE is one such example.)
- The remaining characters can be letters, numbers, hyphens, underscores, or circumflexes (^).

Some file-system procedures, such as DEFINEADD and DEFINEDELETE, require a DEFINE name to be presented as a fixed-length, 24-byte string; for these procedures, the DEFINE name must be left-justified and padded with blank characters up to a total length of 24 characters. Other procedures, such as FILE_OPEN_, take the DEFINE name as a variable-length string that must not contain any blank characters.

Uppercase and lowercase letters in a DEFINE name are equivalent. For example, the name =MY^DEFINE is equivalent to =My^Define.

To refer to a DEFINE, you use the DEFINE name in your program. Where you use the name depends on which class of DEFINE you are using:

- You can use a CLASS CATALOG DEFINE name anywhere that you can use an SQL catalog name. Refer to the *Guardian User's Guide* and the *NonStop SQL/MP Reference Manual* for detailed information about the CLASS CATALOG DEFINE and its attributes.
- You can use a CLASS MAP DEFINE name anywhere that you can use a file name. Refer to the *Guardian User's Guide* and the *TACL Reference Manual* for detailed information about the CLASS MAP DEFINE and its attributes.
- You can use a CLASS TAPE DEFINE name anywhere you can specify the name of a tape file. Refer to the *Guardian Disk and Tape Utilities Reference Manual* for detailed information about the CLASS TAPE DEFINEs and their attributes.
- You can use a CLASS TAPECATALOG DEFINE name anywhere you can specify the name of a labeled tape file. It is used in place of a CLASS TAPE DEFINE, adding several attributes for control of cataloging files that are read from and written to tape. Refer to the *DSM/Tape Catalog User's Guide* for detailed information about the CLASS TAPECATALOG DEFINEs and their attributes.
- You can use a CLASS SPOOL DEFINE name anywhere that you can specify the name of a spooler collector. Refer to the *Spooler Utilites Reference Manual* and

the *Spooler Plus Utilites Reference Manual* for detailed information about the CLASS SPOOL DEFINE and its attributes.

- You can use a CLASS SEARCH DEFINE name as a parameter to the FILENAME_RESOLVE_ procedure described in Section 13, Manipulating File Names.
- You can use a CLASS SORT DEFINE name to specify sort parameters interactively using the RUN FASTSORT command or programmatically as a parameter to the SORTBUILDPARAM procedure. CLASS SUBSORT DEFINEs are referred to from a CLASS SORT DEFINE for passing additional parameters. Refer to the *FastSort Manual* for detailed information about CLASS SUBSORT DEFINE and CLASS SORT DEFINE and their attributes.

DEFINE Attributes

A set of attributes is associated with each DEFINE. One attribute associated with every DEFINE is the CLASS attribute. The CLASS attribute determines which other attributes can be associated with the DEFINE.

Each attribute has:

- An attribute name that you cannot change.
- A data type that determines the kind of value you can assign to the attribute.
- A value that you assign using procedure calls. Some attributes have default values.

The following paragraphs describe the possible data types that a DEFINE attribute can have and the values those attributes can take. For a complete description of each CLASS DEFINE and each possible attribute, refer to the specific manuals listed under DEFINE Names on page 7-7.

Attribute Data Types

When you assign a value to an attribute (using the DEFINESETATTR procedure), you specify the value as a parameter to the procedure call. This parameter must be declared as type STRING in your program and contain ASCII characters; even if the attribute is a number, you must pass the number in ASCII representation.

The STRING values that you can specify for a particular DEFINE attribute are determined by the data type of the DEFINE attribute. When the DEFINESETATTR procedure is called to assign an attribute value, the system verifies that the STRING value you specified matches the data type of the attribute.

The available attribute data types are:

string	The attribute can contain a string of ASCII characters.
number	The attribute can contain an integer. This integer can be preceded by a plus or minus sign. It must not contain a decimal point.

filename	The attribute can contain a file name. The file name can be fully or partially qualified. A partially qualified file name is expanded by the file system using the volume and subvolume that you specify as a parameter to the DEFINESETATTR procedure.
subvolname	The attribute can contain a subvolume name. The subvolume name can be fully or partially qualified. A partially qualified subvolume name is expanded by the file system using the volume that you specify as a parameter to the DEFINESETATTR procedure. If no such volume is specified, then the value provided by the =_DEFAULTS DEFINE is used.
volname	The attribute can contain a volume name, which can be fully or partially qualified. A partially qualified volume name is expanded by the file system using the node name specified in the =_DEFAULTS DEFINE.
keyword	The attribute can contain one of a predefined set of keywords. These keywords are specific to the particular DEFINE class such as SPOOL or TAPE.

Attribute Values

You can assign values to each attribute of a DEFINE using procedure calls.

Three kinds of attributes exist: defaulted, required, and optional. Defaulted attributes are assigned a default value by the file system; this default value is used if you do not assign another value to the attribute.

You must use procedure calls to assign values to required attributes. If you do not assign a value to a required attribute, an error occurs when you attempt to add the DEFINE to the context of a process (or if you check the consistency of the working set using the DEFINEVALIDATEWORK procedure).

Optional attributes do not have default values and are not required.

Some attributes can be assigned a list of values, rather than a single value. A list of values is specified as follows:

```
(value1,value2[,value3] ...  [,valuen])
```

The attribute value you specify must be consistent. For example, some DEFINEs include attributes that are mutually exclusive. The system checks attribute values for consistency and completeness.

CLASS Attribute

All DEFINEs have one special attribute called the CLASS attribute. The CLASS attribute specifies whether the DEFINE is a TAPE, TAPECATALOG, SPOOL, MAP, SEARCH, SORT, SUBSORT, CATALOG, or DEFAULTS CLASS DEFINE.

The CLASS attribute determines what other attributes are associated with the DEFINE.

The CLASS attribute has a data type of keyword. When assigning values to DEFINE attributes, you must assign one of these values to the CLASS attribute first. Assigning a value to the CLASS attribute causes default values to be assigned to other attributes in that DEFINE CLASS.

The attributes of a particular DEFINE are distinct from attributes of the other DEFINE classes, even when the attributes have the same names.

Working With DEFINEs

The most common use of DEFINEs is to pass information from a parent process to its child processes. For processes that are created by TACL, you can create DEFINEs using SET DEFINE commands and ADD DEFINE commands; see the *Guardian User's Guide*. Other processes can pass DEFINEs received from their creators down to their child processes, or they can add DEFINEs using procedure calls such as DEFINESETATTR and DEFINEADD; see Adding DEFINEs later in this section.

See Section 16, Creating and Managing Processes, for further information about propagating DEFINEs, including how to prevent DEFINEs from propagating.

As the recipient of DEFINE information, all you need to do is:

1. Make sure that DEFINE mode is turned on.
2. Refer to the DEFINE by name to make use of the attributes specified in the DEFINE.

Enabling DEFINEs

Every process has an attribute called DEFINE mode. DEFINE mode controls whether DEFINEs can be used or created. The DEFINE mode of a process can be on or off.

- When DEFINE mode is off, the DEFINEs that exist in the context of the process are ignored. When your process starts another process, the DEFINEs in the context of your current process are not propagated to the new process unless you specify otherwise in a parameter to the PROCESS_CREATE_ procedure (see Section 16, Creating and Managing Processes, for details of propagating DEFINEs to other processes). The one exception to this rule is the =_DEFAULTS DEFINE, which is always propagated to the child process.

  You cannot add or replace DEFINEs in the context of the current process when DEFINE mode is off.

- When DEFINE mode is on, processes use the DEFINEs that exist in the context of the current process. When your process starts another process, the DEFINEs in the context of the current process are propagated to the new process. You can add or replace DEFINEs in the context of the current process when DEFINE mode is on.

By default, DEFINE mode is turned on, so it is normally not necessary to enable DEFINE mode. If DEFINE mode is not enabled, however, you can turn it on as described below.

Note. DEFINE mode, like DEFINEs themselves, is propagated from the parent process. If DEFINE mode is turned on in the creator process, then it is also turned on in the child process, unless the parent process specifically requested something else. If DEFINE mode is turned off in the parent process, then, by default, it is also turned off in the child process. Therefore, if DEFINE mode is turned off for your process, you should find out why before proceeding. The process that turned it off might have had a specific reason to do so.

You enable DEFINE mode for the current process by calling the DEFINEMODE procedure with the first parameter set to 1 as follows:

```
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR <> 0 THEN ...
```

The previous setting of the DEFINE mode is returned in OLD^VALUE: 0 indicates DEFINE mode was turned off, and 1 indicates DEFINE mode was turned on. The procedure returns error 2067 if the supplied new value is illegal.

To disable DEFINE mode again, simply call the DEFINEMODE procedure again but with NEW^VALUE set to 0:

```
NEW^VALUE := 0;
CALL DEFINEMODE(NEW^VALUE,
                OLD^VALUE);
```

You can also enable or disable DEFINE mode in a process your program is creating by use of the *`create-options`* parameter of the PROCESS_CREATE_ procedure. Section 16, Creating and Managing Processes, provides details.

Referring to DEFINEs

Assuming DEFINE mode is already on, you simply refer to a DEFINE by name in order to use it.

Adding DEFINEs

This subsection discusses the system procedures that permit you to enable DEFINEs and add DEFINEs to the context of the current process. Assuming DEFINE mode is already on, you perform the following steps when processing DEFINEs:

1. Set the attributes of a new DEFINE in the working set of your process and check the working set for errors
2. Add/replace the DEFINE in the context of your process

These operations are described in the following paragraphs, as well as how to delete a DEFINE from the process context and how to save and restore DEFINEs.

Setting Attributes in the Working Set

The primary method for setting attributes in the working set is the DEFINESETATTR procedure. The DEFINESETLIKE and DEFINERESTORE procedures, however, also have the effect of setting working set attributes.

The following paragraphs describe each of these procedures.

Setting Attributes Using the DEFINESETATTR Procedure

The DEFINESETATTR procedure assigns a value to an individual attribute in the working set.

When assigning a value to any attribute, you must supply DEFINESETATTR with the name of the attribute you want to assign a value to, the actual attribute value, and the length of the value string.

In addition to the name, value, and length of the attribute, if the attribute type is filename, subvolname, or volname, you should also supply the *`default-names`* parameter to supply values for the node name, volume name, and subvolume name to be used to convert the attribute value into an internal form. A convenient way of obtaining these values is by reading them from the Startup message; Section 8, Communicating With a TACL Process, describes how to do this.

Assign a value to the CLASS attribute first. When you do this, the file system initializes all defaulted attributes for that DEFINE CLASS with default values. The following example assigns the value “MAP” to the CLASS attribute:

```
DEFAULT^VALUES ':=' "$VOL    SUBVOL  ";
 .
 .
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "MAP" -> @S^PTR;
ATTRIBUTE^LENGTH := @S^PTR '-' @ATTRIBUTE^VALUE;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH);
IF ERROR <> 0 THEN ...
```

For a complete list of the error conditions that this procedure might return, see the *Guardian Procedure Calls Reference Manual*.

After assigning a value to the CLASS attribute, use additional DEFINESETATTR calls to assign values to other attributes:

```
ATTRIBUTE^NAME ':=' "FILE            ";
ATTRIBUTE^VALUE ':=' "MYFILE" -> @S^PTR;
ATTRIBUTE^LENGTH := @S^PTR '-' @ATTRIBUTE^VALUE;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LENGTH,
                       DEFAULT^VALUES);
IF ERROR <> 0 THEN ...
```

When you have assigned a value to an attribute, the system verifies that the value you specified matches the data type of the attribute. The procedure returns an error if the value does not match the data type.

Setting Attributes Using the DEFINESETLIKE Procedure

For any DEFINE that already exists in the context of the current process, you can initialize the attributes of the working set to the values of the attributes of that DEFINE. You do this using the DEFINESETLIKE procedure:

```
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINESETLIKE(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

After initializing the working set attributes using this procedure, you can call DEFINESETATTR to alter the values of specific attributes.

Setting Attributes Using the DEFINERESTORE Procedure

If you have saved a working set or a DEFINE using the DEFINESAVE procedure (see Saving and Restoring DEFINEs, later in this section), you can initialize the attributes of the working set by restoring that DEFINE or saved working set using the DEFINERESTORE procedure:

```
LITERAL RESTORE^SAVED = 2;
 .
 .
ERROR := DEFINERESTORE(BUFFER,
                       RESTORE^SAVED);
```

Setting the second parameter to 2 forces the saved DEFINE to be restored as the working set.

Checking the Working Set for Errors

After assigning values to the attributes in the working set, you can check the attribute values for errors using the DEFINEVALIDATEWORK procedure. This procedure indicates whether the attributes in the working set are incomplete, inconsistent, or invalid.

```
ERROR := DEFINEVALIDATEWORK(CHECK^NUM);
CASE ERROR OF
BEGIN
   0 ->         !no error
   2057 ->      !incomplete
   2058 ->      !inconsistent
   2059 ->      !invalid
   OTHERWISE -> !some other problem
END;
```

Here, the program checks whether the ERROR value returned is 2057, 2058, or 2059.

- 2057 indicates that the working set is incomplete: that is, a required parameter for the DEFINE CLASS is missing.
- 2058 indicates that the working set is inconsistent; CHECK^NUM qualifies the inconsistency. For more information about CHECK^NUM, refer to the *Guardian Procedure Errors and Messages Manual.*
- 2059 indicates that the working set is invalid.

The system also checks the assigned attributes when you call the DEFINEADD procedure to add the DEFINE to the context of your process.

Adding a DEFINE to the Context of Your Process

After setting the attributes in the working set, you can assign a DEFINE name to that attribute set and save it in the context of your process. You do this using the DEFINEADD procedure:

```
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINEADD(DEFINE^NAME,
                   !replace!,
                   CHECK^NUM);
CASE ERROR OF
BEGIN
   0 ->         !no error
   2049 ->      !syntax error in name
   2050 ->      !DEFINE already exists
   2051 ->      !DEFINE does not exist
   2052 ->      !can’t get file-system buffer space
   2053 ->      !can’t get physical memory
   2054 ->      !bounds error
   2057 ->      !incomplete
   2058 ->      !inconsistent
   2059 ->      !invalid
   2066 ->      !missing parameter
   2069 ->      !DEFINE type not permitted
   OTHERWISE -> !some other problem
END;
```

When you add a DEFINE to the context of your process, the system checks the values that you specified to ensure that they are consistent, complete, and valid. If they are not, a nonzero value is returned. If the error number returned is equal to 2058, then the system is reporting an inconsistency. The inconsistency is qualified in CHECK^NUM. The *Guardian Procedure Errors and Messages Manual* describes each error.

You can also use the DEFINEADD procedure to replace a DEFINE by the same name in the context of your current process. To do this, you need to set the second parameter of the DEFINEADD call to 1.

```
LITERAL REPLACE = 1;
 .
 .
DEFINE^NAME ':=' "=TAPE1                  ";
```

```
ERROR := DEFINEADD(DEFINE^NAME,REPLACE,CHECK^NUM);
CASE ERROR OF
BEGIN
    .
    .
```

Deleting DEFINEs From the Process Context

You can delete DEFINEs from the context of a process using the DEFINEDELETE or DEFINEDELETEALL procedure. The DEFINEDELETE procedure deletes a specific DEFINE, for example:

```
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINEDELETE(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

The DEFINEDELETEALL procedure deletes all DEFINEs from the context of the current process except the =_DEFAULTS DEFINE. (You cannot delete the =_DEFAULTS DEFINE.)

```
CALL DEFINEDELETEALL;
```

Saving and Restoring DEFINEs

Sometimes it can be useful to maintain several independent contexts for starting new processes. The DEFINESAVE and DEFINERESTORE procedures provide a way to save a DEFINE or a working set of attributes in a memory buffer and then restore the contents of that buffer into a new DEFINE or working set.

You save a DEFINE in a buffer using the DEFINESAVE procedure:

```
BUFF^LEN := 500;
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINESAVE(DEFINE^NAME,
                    TAPE^BUFFER,
                    BUFF^LEN,
                    DEFINE^LEN);
IF ERROR <> 0 THEN ...
```

The above example saves the =TAPE1 DEFINE in a 500-byte buffer named TAPE^BUFFER. The actual length of the DEFINE is returned in DEFINE^LEN. The final parameter is set to 0 to tell the system to save the DEFINE named in the first parameter. When DEFINESAVE saves a DEFINE in a user buffer, the DEFINE is saved in another format. You should not attempt to modify this format, otherwise DEFINERESTORE may not be able to restore the DEFINE.

You restore a DEFINE using the DEFINERESTORE procedure. The DEFINERESTORE procedure can add a DEFINE to the process context or change the attributes of an existing DEFINE. However, you must specify whether you intend to add or alter the DEFINE.

```
LITERAL ADD^DEFINE = 0,
        CHG^DEFINE = 1;
.
.
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINERESTORE(TAPE^BUFFER,
                       ADD^DEFINE,
                       DEFINE^NAME,
                       CHECK^NUM);
IF ERROR = 2058 THEN ...
ELSE IF ERROR <> 0 THEN ...

DEFINE^NAME ':=' "=TAPE2                  ";
ERROR := DEFINERESTORE(TAPE^BUFFER2,
                       CHG^DEFINE,
                       DEFINE^NAME,
                       CHECK^NUM);
IF ERROR = 2058 THEN ...
ELSE IF ERROR <> 0 THEN ...
```

The first of the two calls shown above adds the DEFINE to the context of the current process. You specify that the DEFINE is to be added by setting the second parameter to 0. If a DEFINE of the name =TAPE1 already exists, then the system returns an error.

The second call above changes the value of an existing DEFINE. Here, the second parameter is set to 1. If the =TAPE2 DEFINE does not already exist in the context of the current process, then the system returns an error.

In addition to returning a standard error number, if the system detects that the saved DEFINE (or saved working set) is inconsistent, then an additional error value is returned in CHECK^NUM to give more information about the error.

Saving and Restoring the Working Set

You can save the working set of attributes in the background and have the option to restore the saved working set later. There are three areas where sets of attributes can be held outside of DEFINEs. The three places are the working set and two background areas.

- The DEFINESAVEWORK and DEFINERESTOREWORK procedures move sets of attributes between the working set and one background area.
- The DEFINESAVEWORK2 and DEFINERESTOREWORK2 procedures move sets of attributes between the working set and the other background area.

You cannot use DEFINERESTOREWORK2 to restore a working set saved by DEFINESAVEWORK, nor can you use DEFINERESTOREWORK to restore a working set saved by DEFINESAVEWORK2.

The following example uses DEFINESAVEWORK2 to save the attribute values stored in the current working set. This procedure copies the attributes in the working set to the background set:

```
ERROR := DEFINESAVEWORK2;
IF ERROR <> 0 THEN ...
```

Any attributes that were stored in the background set are destroyed.

The next example restores the background set into the working set using the DEFINERESTOREWORK2 procedure:

```
ERROR := DEFINERESTOREWORK2;
IF ERROR <> 0 THEN ...
```

Note that there is actually a third way of saving and restoring a working set that uses a buffer instead of a background working set. This method uses the DEFINESAVE and DEFINERESTORE procedures as already described.

Using DEFINEs: An Example

The sample program shown in this subsection allows the user to create DEFINEs interactively before starting a process that uses the DEFINEs.

This program first prompts the user for the CLASS of the DEFINE that the user needs to create. The CREATE^DEFINES procedure then calls a procedure that depends on the DEFINE CLASS. For example, if the user selected CLASS TAPE, then the SET^TAPE procedure gets called. This procedure then prompts the user for a value for each of the DEFINE attributes that pertain to the selected type.

When the user has responded to each possible attribute, control returns to the CREATE^DEFINES procedure, which calls DEFINEVALIDATEWORK to check that the working set is consistent before prompting the user for a DEFINE name. When the user has entered a valid DEFINE name, the CREATE^DEFINES procedure creates the DEFINE.

Finally, control returns to the MAIN procedure, which prompts the user to create another DEFINE. When the user declines to create any more DEFINEs, the MAIN procedure prompts the user for the name of the program file to execute and then executes the program.

This sample program allows the user to create the following classes of DEFINEs:

- CLASS MAP DEFINEs
- CLASS SEARCH DEFINEs
- CLASS TAPE DEFINEs
- CLASS SPOOL DEFINEs
- CLASS SORT DEFINEs
- CLASS SUBSORT DEFINEs

To add another class of DEFINE, you need to add a procedure, similar to the SET^TAPE procedure, that prompts the user to enter attribute values for the new DEFINE CLASS. You will also need to make changes to the GET^DEFINE^CLASS and CREATE^DEFINES procedures as indicated in the following pages.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;   !maximum file-name
                                           ! length
LITERAL BUFSIZE = 512;
LITERAL ABEND   = 1;

STRING .SBUFFER[0:BUFSIZE];       !I/O buffer (one extra char)
STRING .S^PTR;                    !pointer to end of string
INT     LOGNUM;                   !log file number
INT     TERMNUM;                  !terminal file number

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  PROCESS_GETINFO_,FILE_OPEN_,WRITEREADX,WRITEX,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN,DEFINESETATTR,DEFINEADD,
?  DEFINEVALIDATEWORK,PROCESS_CREATE_)
?LIST
```

```
!---------------------------------------------------------
! Here are some DEFINEs to make it easier to format and
! print messages.
!----------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE (SBUFFER, 0) #;


!  Print a string:

    DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                     PUT^STR (S);
                                     PRINT^LINE;  END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there isn't a file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_(!process^handle!,
                      !specifier!,
                      ABEND);
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to do the display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT         FNUM;
BEGIN
   INT         ERROR;
   STRING     .FNAME[0:MAXFLEN - 1];
   INT         FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!-----------------------------------------------------------
! Procedure to print DEFINE errors on the terminal.
!-----------------------------------------------------------

PROC DEFINE^ERRORS(ERR^NUM);
INT   ERR^NUM;
BEGIN

!  Display the error number:

   START^LINE;
   PUT^STR("DEFINE error ");
   PUT^INT(ERR^NUM);
   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Put the error text in the output buffer:

   START^LINE;
   CASE ERR^NUM OF
   BEGIN
      2049 -> PUT^STR("A syntax error occurred in name");
      2050 -> PUT^STR("DEFINE already exists");
      2051 -> PUT^STR("DEFINE does not exist");
      2052 -> PUT^STR
        ("Not enough buffer space or PFS allocation failed");
      2053 -> PUT^STR("Unable to obtain physical memory");
      2054 -> PUT^STR("Bounds error on parameter");
      2055 -> PUT^STR("Attribute not supported");
      2057 -> PUT^STR("DEFINE or working set incomplete");
      2058 -> PUT^STR
        ("DEFINE or working set is not consistent");
      2059 -> PUT^STR("DEFINE or working set is invalid");
      2060 -> PUT^STR("No more DEFINEs");
      2061 -> PUT^STR("No more attributes");
      2062 -> PUT^STR("Attribute name too long");
      2063 -> PUT^STR
        ("A syntax error occurred in DEFAULT^NAMES");
      2064 -> PUT^STR("The attribute cannot be reset");
      2066 -> PUT^STR("Missing parameter");
      2067 -> PUT^STR
        ("Attribute contained an illegal value");
      2068 -> PUT^STR("Saved DEFINE was of invalid CLASS");
      2069 -> PUT^STR
        ("The DEFINE mode of the process does not "
         & "permit the addition of the DEFINE type");
      2075 -> PUT^STR("Invalid options parameter");
      2076 -> PUT^STR("User's buffer is too small");
      2077 -> PUT^STR
        ("Buffer or DEFINE name is an invalid segment");
      2078 -> PUT^STR
        ("Buffer does not contain a valid saved DEFINE");
      OTHERWISE -> PUT^STR("Error number unrecognized");
   END;
```

```
!  Display the error description on the terminal:

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

END;


!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! This procedure asks the user for the CLASS of the next
! DEFINE:
!
! "1" for a CLASS MAP DEFINE
! "2" for a CLASS SEARCH DEFINE
! "3" for a CLASS TAPE DEFINE
! "4" for a CLASS SPOOL DEFINE
! "5" for a CLASS SORT DEFINE
! "6" for a CLASS SUBSORT DEFINE
! "7" for a CLASS CARALOG DEFINE
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^DEFINE^CLASS;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the DEFINE CLASS:

   PRINT^BLANK;
   PRINT^STR("Type '1' for a CLASS MAP DEFINE, ");
   PRINT^STR("     '2' for a CLASS SEARCH DEFINE, ");
   PRINT^STR("     '3' for a CLASS TAPE DEFINE, ");
   PRINT^STR("     '4' for a CLASS SPOOL DEFINE,  ");
   PRINT^STR("     '5' for a CLASS SORT DEFINE,  ");
   PRINT^STR("     '6' for a CLASS SUBSORT DEFINE,  ");
   PRINT^STR("     '7' for a CLASS CATALOG DEFINE,  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!-----------------------------------------------------------
! Procedure to prompt the user for a DEFINE attribute and
! call DEFINESETATTR if the user provides the attribute.
!-----------------------------------------------------------

PROC DEFINE^ATTR(ATTR^NAME,VALUES^LIST,VALUES^LEN);
STRING   .ATTR^NAME;
STRING   .VALUES^LIST;
INT       VALUES^LEN;

BEGIN
   INT      .DEFAULT^NAMES[0:7] := "$APPLS  PROGS   ";
   INT       COUNT^READ;
   INT       ERROR;

!  Obtain a value for the attribute from the user:

DO^AGAIN:
   PRINT^BLANK;
   PRINT^STR
     ("Enter a value for the attribute " & ATTR^NAME FOR 16);
   PRINT^STR
     ("Possible values are: " & VALUES^LIST FOR VALUES^LEN);
   PRINT^BLANK;
   SBUFFER ':=' "Choice:  " ->@S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   IF COUNT^READ <> 0 THEN
   BEGIN
      ERROR := DEFINESETATTR(ATTR^NAME,SBUFFER,COUNT^READ,
                             DEFAULT^NAMES);
      IF ERROR <> 0 THEN
      BEGIN
         CALL DEFINE^ERRORS(ERROR);
         GOTO DO^AGAIN;
      END;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! MAP DEFINE.
!-------------------------------------------------------------

PROC SET^MAP;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be MAP" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "FILE            ";
   VALUES^LIST ':=' "Any valid file name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! SEARCH DEFINE.
!-------------------------------------------------------------

PROC SET^SEARCH;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];
   INT I := 0;

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be SEARCH" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   WHILE I < 21 DO
   BEGIN
      NAME[0] ':=' " ";
      NAME[1] ':=' NAME[0] FOR 15 BYTES;
      START^LINE;
      PUT^STR("SUBVOL");
      PUT^INT(I);
      NAME[0] ':=' SBUFFER FOR (@S^PTR '-' @SBUFFER) BYTES;
      VALUES^LIST ':='
         "One or more subvolumes or CLASS DEFAULTS DEFINES"
         -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);
      I := I + 1;
   END;

   I := 0;
   WHILE I < 21 DO
   BEGIN
      NAME[0] ':=' " ";
      NAME[1] ':=' NAME[0] FOR 15 BYTES;
      START^LINE;
      PUT^STR("RELSUBVOL");
      PUT^INT(I);
      NAME[0] ':=' SBUFFER FOR (@S^PTR '-' @SBUFFER) BYTES;
      VALUES^LIST ':='
         "One or more subvolumes or CLASS DEFAULTS DEFINES"
         -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! TAPE DEFINE.
!------------------------------------------------------------

PROC SET^TAPE;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be TAPE" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "BLOCKLEN        ";
   VALUES^LIST ':=' "Any valid block length" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "DENSITY         ";
   VALUES^LIST ':=' "800, 1600, or 6250 bpi" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "DEVICE          ";
   VALUES^LIST ':=' "A valid tape device name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "EBCDIC          ";
   VALUES^LIST ':=' "IN, OUT, ON, or OFF" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "EXPIRATION      ";
   VALUES^LIST ':='
       "Any valid date (month day year)" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "FILEID          ";
   VALUES^LIST ':=' "Any valid tape file name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "FILESECT        ";
   VALUES^LIST ':=' "A number in the range 0001 through 9999"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
```

```
NAME ':=' "FILESEQ         ";
VALUES^LIST ':=' "A number in the range 0001 through 9999"
                 -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "GEN             ";
VALUES^LIST ':=' "A number in the range 0001 through 9999"
                 -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "LABELS          ";
VALUES^LIST ':=' "ANSI, IBM, OMITTED, or BYPASS"
                 -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "MOUNTMSG        ";
VALUES^LIST ':=' "Any text string" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "OWNER           ";
VALUES^LIST ':=' "Any valid owner ID" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "RECFORM         ";
VALUES^LIST ':=' "F or U" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "RECLEN          ";
VALUES^LIST ':=' "A valid record length" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "REELS           ";
VALUES^LIST ':=' "A number in the range 1 through 255"
                 -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "RETENTION       ";
VALUES^LIST ':=' "Any integer value" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);

NAME ':=' "SYSTEM          ";
VALUES^LIST ':=' "A valid system name" -> @S^PTR;
CALL DEFINE^ATTR(NAME,VALUES^LIST,
                 @S^PTR '-' @VALUES^LIST);
```

```
   NAME ':=' "USE             ";
   VALUES^LIST ':=' "IN, OUT, EXTEND, or OPENFLAG" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "VERSION         ";
   VALUES^LIST ':=' "A number in the range 00 through 99"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "VOLUME          ";
   VALUES^LIST ':=' "A six-byte volume ID or SCRATCH"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
END;
```

```
!------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! SPOOL DEFINE.
!------------------------------------------------------------

PROC SET^SPOOL;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be SPOOL" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "BATCHID         ";
   VALUES^LIST ':=' "A valid job ID" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "BATCHNAME       ";
   VALUES^LIST ':=' "A 1 to 31 character batch name"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "COPIES          ";
   VALUES^LIST ':=' "A number in the range 1 through 32767"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "FORM            ";
   VALUES^LIST ':=' "A 1 to 16 character form name"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "HOLD            ";
   VALUES^LIST ':=' "ON or OFF" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "HOLDAFTER       ";
   VALUES^LIST ':=' "ON or OFF" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "LOC             ";
   VALUES^LIST ':=' "A valid spooler location" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
   NAME ':=' "MAXPRINTLINES   ";
   VALUES^LIST ':=' "A number in the range 1 through 65534"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "MAXPRINTPAGES   ";
   VALUES^LIST ':=' "A number in the range 1 through 65534"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "OWNER           ";
   VALUES^LIST ':=' "Any valid owner ID" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "PAGESIZE        ";
   VALUES^LIST ':=' "A number in the range 1 through 32767"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "REPORT          ";
   VALUES^LIST ':=' "A 1 to 16 character report name"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "SELPRI          ";
   VALUES^LIST ':=' "A number in the range 0 through 7"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! SORT DEFINE.
!-------------------------------------------------------------

PROC SET^SORT;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be SORT" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "BLOCK           ";
   VALUES^LIST ':=' "Any multiple of 512 up to 30K"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "CPU             ";
   VALUES^LIST ':=' "A number in the range 0 through 15"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "CPUS            ";
   VALUES^LIST ':=' ["A comma-separated list of numbers ",
                     "in the range 0 through 15"]
      -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "MODE            ";
   VALUES^LIST ':=' "AUTOMATIC, MINSPACE or MINTIME"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "NOTCPUS         ";
   VALUES^LIST ':=' ["A comma-separated list of numbers ",
                     "in the range 0 through 15"]
      -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "PRI             ";
   VALUES^LIST ':=' "A number in the range 1 to 199"
                    -> @S^PTR;

   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
      NAME ':=' "PROGRAM         ";
      VALUES^LIST ':=' "Any valid program file name"
                       -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);

      NAME ':=' "SCRATCH         ";
      VALUES^LIST ':=' "Any valid disk volume name" -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);

      NAME ':=' "SEGMENT         ";
      VALUES^LIST ':=' "A number 64 or greater" -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);

      NAME ':=' "SUBSORTS        ";
      VALUES^LIST ':='
             "A comma-separated list of CLASS SUBSORT DEFINES"
             -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);

      NAME ':=' "SWAP            ";
      VALUES^LIST ':='
          "Any valid local file or local volume name" -> @S^PTR;
      CALL DEFINE^ATTR(NAME,VALUES^LIST,
                       @S^PTR '-' @VALUES^LIST);
   END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! SUBSORT DEFINE.
!-------------------------------------------------------------

PROC SET^SUBSORT;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be SUBSORT" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "BLOCK          ";
   VALUES^LIST ':=' "Any multiple of 512 up to 30K"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "CPU             ";
   VALUES^LIST ':=' "A number in the range 0 through 15"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "PRI             ";
   VALUES^LIST ':=' "A number in the range 1 through 199"
                    -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "PROGRAM         ";
   VALUES^LIST ':=' "A valid program file name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "SCRATCH         ";
   VALUES^LIST ':='
              "A valid unstructured file name or volume name"
              -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "SEGMENT         ";
   VALUES^LIST ':=' "A number 64 or greater" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "SWAP            ";
   VALUES^LIST ':=' "A valid file or volume name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
END;
```

```
!------------------------------------------------------------
! Procedure to prompt the user for all attributes for a CLASS
! CATALOG DEFINE.
!------------------------------------------------------------

PROC SET^CATALOG;
BEGIN
   STRING .VALUES^LIST[0:BUFSIZE - 1];
   STRING .NAME[0:15];

   NAME ':=' "CLASS           ";
   VALUES^LIST ':=' "Must be CATALOG" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);

   NAME ':=' "SUBVOL          ";
   VALUES^LIST ':=' "Any valid subvolume name" -> @S^PTR;
   CALL DEFINE^ATTR(NAME,VALUES^LIST,
                    @S^PTR '-' @VALUES^LIST);
END;


!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than a number
! in the range 1 through 7.
!------------------------------------------------------------

PROC INVALID^SELECTION;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and
!  then return to prompt again for a valid function:

   PRINT^STR("INVALID SELECTION: "  &
             "Type number in range 1 through 7.  ");
END;
```

```
!-----------------------------------------------------------
! Procedure to create DEFINEs.  The procedure prompts the
! user for the DEFINE CLASS, then for each attribute that a
! DEFINE of the selected CLASS can have.  When the attributes
! have all been entered, the procedure checks their validity
! and creates the DEFINE.
!-----------------------------------------------------------

PROC CREATE^DEFINES;
BEGIN
   STRING  .DEFINE^NAME[0:23];
   STRING   CMD;
   INT      ERROR;
   INT      COUNT^READ;

!  Loop until all DEFINEs have been created:

   DO BEGIN

   !  Prompt user for CLASS of DEFINE required

      CMD := GET^DEFINE^CLASS;

   !  Call the appropriate procedure to prompt for the
   !  attributes for a DEFINE of the selected CLASS.
   !  Repeat if error on validation check.

   DO^AGAIN:
      CASE CMD OF
      BEGIN
         "1" -> CALL SET^MAP;

         "2" -> CALL SET^SEARCH;

         "3" -> CALL SET^TAPE;

         "4" -> CALL SET^SPOOL;

         "5" -> CALL SET^SORT;

         "6" -> CALL SET^SUBSORT;

         "7" -> CALL SET^CATALOG;

         OTHERWISE -> CALL INVALID^SELECTION;
      END;

   !  Check the working set for errors.  If errors, have
   !  option to continue or stop.

      ERROR := DEFINEVALIDATEWORK;
      IF ERROR <> 0 THEN
```

```
        BEGIN
           CALL DEFINE^ERRORS(ERROR);
           GOTO DO^AGAIN;
        END;

     !  Prompt the user for the DEFINE name:

     REENTER^NAME:
        PRINT^BLANK;
        SBUFFER ':=' "Please Enter a Name for the DEFINE: "
                     ->@S^PTR;
        CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                        BUFSIZE,COUNT^READ);
        IF <> THEN CALL FILE^ERRORS(TERMNUM);
        IF SBUFFER[0] <> "=" THEN
        BEGIN
           PRINT^STR("First Character Must Be '='");
           GOTO REENTER^NAME;
        END;
        IF COUNT^READ > 24 THEN
        BEGIN
           PRINT^STR
              ("Maximum DEFINE name length 24 characters");
           GOTO REENTER^NAME;
        END;

        DEFINE^NAME[0] ':=' " ";
        DEFINE^NAME[1] ':=' DEFINE^NAME[0] FOR 23;
        DEFINE^NAME[0] ':=' SBUFFER FOR COUNT^READ;

     !  Add the DEFINE to the PFS:

        ERROR := DEFINEADD(DEFINE^NAME);
        IF ERROR <> 0 THEN
        BEGIN
           CALL DEFINE^ERRORS(ERROR);
           GOTO DO^AGAIN;
        END;

     !  Prompt the user to enter more DEFINEs:

        PRINT^BLANK;
        SBUFFER ':=' "Do You Wish to Enter More DEFINEs (y/n)?"
                     -> @S^PTR;
        CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                        BUFSIZE,COUNT^READ);
        IF <> THEN CALL FILE^ERRORS(TERMNUM);
     END
     UNTIL SBUFFER[0] <> "y" AND SBUFFER[0] <> "Y";
  END;
```

```
!------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .TERMNAME[0:MAXFLEN - 1];     !terminal file
   INT     TERMLEN;
   INT     ERROR;

!  Read and discard startup messages:

   CALL INITIALIZER;

!  Open the terminal file. For simplicity we use the home
!  terminal.  The recommended approach is to use the IN file
!  read from the Startup message; see Section 8 for
!  details:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERMNAME:MAXFLEN,
                         TERMLEN);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
END;
```

```
!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize, then it goes into a loop calling GET^COMMAND
! to get the next user request and calling the procedure
! to carry out that request.
!-------------------------------------------------------------

PROC STARTER MAIN;
BEGIN
   STRING    CMD;
   INT       COUNT^READ;
   INT       ERROR;

   CALL INIT;

!  Prompt user to request DEFINEs:

   SBUFFER ':=' "Do You Want to Set up DEFINEs?  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Call CREATE^DEFINES if the user responds 'y':

   IF SBUFFER[0] = "y" OR SBUFFER[0] = "Y" THEN
   CALL CREATE^DEFINES;

!  Prompt the user for the program file name to execute:

   PRINT^BLANK;
   SBUFFER ':=' "Please enter program file name:  "
                -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Execute the program:

   ERROR := PROCESS_CREATE_(SBUFFER:COUNT^READ);
   IF ERROR <> 0 THEN
   BEGIN
      PRINT^STR("Unable to start specified program");
      CALL PROCESS_STOP_(!process^handle!,
                         !specifier!,
                         ABEND);
   END;
END;
```

8

Communicating With a TACL Process

This section discusses how processes communicate with TACL processes, including:

- How to set up the process environment by using commands at the TACL prompt.
- How the process environment information is passed to the application using command-interpreter messages. These messages include the Startup, Assign, and Param messages. They are part of a different set of messages than the system messages discussed in Section 6, Communicating With Processes. However, the delivery mechanism is the same: the recipient process reads the messages from its $RECEIVE file.
- How a new process receives its process environment information from its $RECEIVE file using the INITIALIZER procedure.
- How a new process can receive its process environment information without using INITIALIZER.
- How a process wakes the TACL process using the Wakeup command-interpreter message.
- How a process sends text for display by the TACL process using the Display command-interpreter message.

To run C or C++ programs in the Guardian environment, see the *C/C++ Programmer's Guide*.

Setting Up the Process Environment

A terminal user uses the TACL process to specify the environment in which a process will run. Some information can be specified before running the process; some information is specified with the RUN command itself. In either case, this information is sent to the new process in one or more interprocess messages.

The following TACL commands can affect the parameter information sent to a new process:

ASSIGN	Makes logical-file assignments. A logical-file assignment equates a file name with a logical file of a program and optionally assigns file characteristics to that file. For each ASSIGN in effect when the program is run, one Assign message is sent to the new process at the option of the new process.
CLEAR	Clears ASSIGN and PARAM settings.
PARAM	Associates an ASCII value with a parameter name. This command is typically used to pass arbitrary string values to the program. If any PARAMs are in effect when a program is run, a single Param message containing all parameter names and values is sent to the new process at the option of the new process.
RUN	Specifies the input and output files and optional parameter string to be passed to the new process. This information, along with the default volume and subvolume names, is passed to the new process in the Startup message.
SYSTEM	When used with a nonblank system name, implicitly causes remote programs to be run. In this case, the volume IN and OUT parameters passed in the Startup message contain the network node number in the upper bytes. That is, the upper byte contains the backslash character (\), and the second byte contains the network node number. Up to six characters then identify the volume (without the leading dollar sign, $).
VOLUME	Specifies the default volume and subvolume names to be passed to the new process.

The VOLUME, RUN, ASSIGN, PARAM, CLEAR, and SYSTEM commands are described in the *TACL Reference Manual*.

Once the process environment is set up and the RUN command started, the parameter information is passed to the new process in the following sequence:

1. The system sends system message number -103 (Open message) to the new process when the TACL process opens the new process. The TACL process is then able to send messages to the new process.
2. The TACL process sends a Startup message (command-interpreter message code -1) to the new process. This message contains the default volume and subvolume names, as well as the input and output file names and optional parameters.
3. If requested by the new process, the TACL process sends Assign messages to the new process (command-interpreter message code -2). These messages contain logical-file assignments made by the ASSIGN command.
4. If requested by the new process and if the TACL process has accepted any PARAM statements, the TACL process sends a Param message to the new

process (command-interpreter message code -3). This message contains ASCII parameter values set up by the PARAM command.

5. The system sends a system message number -104 (Close message) when the TACL process closes the new process. This message indicates that the TACL process has completed its communication with the new process.

The new process can read the above messages from its $RECEIVE file by calling the INITIALIZER procedure. INITIALIZER allows your program to receive these messages without having to deal directly with $RECEIVE. In addition to reading the messages, INITIALIZER calls your procedures to process the Startup, Assign, and Param messages.

DEFINEs are automatically passed to the new process from its parent process when the new process is created. The new process does not have to do anything to get these DEFINEs.

If a process other than a TACL process opens the new process, it must send the same sequence of messages as the TACL process. Section 16, Creating and Managing Processes, provides details.

Obtaining Startup Information

To obtain startup information, your program must read and process the Startup message. This message is usually part of the startup sequence that takes place between a process and the process that it creates. It is sent to every process that is started by the TACL process. The Startup message can be omitted only if both the parent and child processes agree.

The structure of the Startup message is shown below:

Structure of the Startup message:

```
STRUCT CI^STARTUP;
BEGIN
   INT MSGCODE;                  !word 0 –  value -1

   STRUCT DEFAULT;
   BEGIN
      INT VOLUME[0:3];           !word 1 –  default volume
name
      INT SUBVOL[0:3];           !default subvolume name
   END;

   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];           !word 9 –  IN parameter file
      INT SUBVOL[0:3];           !name of RUN command
      INT FNAME[0:3];
   END;

   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];           !word 21 – OUT parameter file
      INT SUBVOL[0:3];           !name of RUN command
      INT FNAME[0:3];
   END;

   STRING PARAM[0:n-1];          !word 33 – parameter string
(if
END;                             !any) of RUN command
```

The maximum length of the Startup message is 596 bytes, including the trailing null characters. The Startup message contains the following information:

- The value -1 in the first word, which identifies the message as the Startup message from the TACL process.
- The default volume and subvolume names provided by the last execution of the VOLUME command. Eight bytes each are provided for the volume and subvolume names. The volume name must be no greater than seven characters long (including the dollar sign) if a remote volume is specified as the default volume.
- The input and output files supplied to the process by the RUN command. Eight bytes each are provided for the volume, subvolume, and file ID parts of the name.

  If the file is on a remote node, then the upper two bytes of the volume name identify the system. The upper byte contains a backslash character (\), and the second byte contains the node number. This leaves only six bytes for the volume name. (The dollar sign, $, is not included in the volume name if the file is remote.) To supply the network version of a file name in the Startup message, the volume name must be no more that seven characters long (including the dollar sign).

The input and output file names in the Startup message have the same structure as file names on a C-series system. To convert these names to D-series format, you can use the OLDFILENAME_TO_FILENAME_ procedure.

- A parameter string containing any parameters supplied to the RUN command. The parameter string contains exactly the same characters as contained in the parameter string to the RUN command, plus a null byte to terminate the string. If the resulting message has an odd number of bytes, a second null is added.

  If no parameter string data is included, under the standard protocol you must append two null bytes to the end of the Startup message.

A sample execution of the VOLUME and RUN commands follows:

```
VOLUME $APPLS.PROGS
RUN MYPROG /IN INFILE,OUT OUTFILE/ PARAM1
```

The corresponding Startup message sent to the application MYPROG is as follows:

	Word								
MSGCODE	0	-1							
DEFAULT.VOLUME	1	"$"	"A"	"P"	"P"	"L"	"S"	" "	" "
DEFAULT.SUBVOL	5	"P"	"R"	"O"	"G"	"S"	" "	" "	" "
INFILE.VOLUME	9	"$"	"A"	"P"	"P"	"L"	"S"	" "	" "
INFILE.SUBVOL	13	"P"	"R"	"O"	"G"	"S"	" "	" "	" "
INFILE.FNAME	17	"I"	"N"	"F"	"I"	"L"	"E"	" "	" "
OUTFILE.VOLUME	21	"$"	"A"	"P"	"P"	"L"	"S"	" "	" "
OUTFILE.SUBVOL	25	"P"	"R"	"O"	"G"	"S"	" "	" "	" "
OUTFILE.FNAME	29	"O"	"U"	"T"	"F"	"I"	"L"	"E"	" "
PARAM	33	"P"	"A"	"R"	"A"	"M"	"1"	null	null

VST040.VSD

Using INITIALIZER to Read the Startup Message

It is up to the application to choose what to do with the information contained in the Startup message. To use the startup information, your program must first read the Startup message from its $RECEIVE file. The INITIALIZER procedure provides a convenient way to do this.

In its simplest form, INITIALIZER reads the Startup message but does not process it:

```
CALL INITIALIZER;
```

The benefit of doing this is to prevent unwanted run-time messages indicating that the Startup message was not read.

If you want to read the Startup message and then process it, you need to specify a user-supplied procedure to do the processing. Call INITIALIZER as follows:

```
CALL INITIALIZER(!rucb!,
                 !return^data!,
                 START^PROC);
```

Here, START^PROC is a user-supplied procedure that processes the Startup message. The INITIALIZER procedure passes the message to the user-supplied procedure. Typically, the user-supplied procedure saves the Startup message in global data and returns. However, it can alternatively use the RETURN^DATA variable to return data from the user-supplied procedure back to the procedure that called INITIALIZER.

The user-supplied procedure must be declared with parameters that match exactly those expected by the INITIALIZER procedure. See below for an example, or refer to the *Guardian Procedure Calls Reference Manual* for a complete description of the required parameters.

Processing the Startup Message

The example given below is made up of a main procedure that reads the Startup message and another procedure that processes it. Their combined actions are summarized as follows:

1. The main procedure calls the INITIALIZER procedure, giving it the name of a procedure to process the Startup message and an array to receive the response.
2. The INITIALIZER procedure passes the Startup message and message length to the START^IT procedure. The message is passed in the MESSAGE parameter, and the message length is passed in the LENGTH parameter.
3. The START^IT procedure saves the Startup message in global data.
4. The START^IT procedure terminates, returning control to the INITIALIZER procedure.
5. The INITIALIZER procedure returns control to the main procedure.

The main procedure can then access the Startup information in the global data area and open the IN and OUT files identified in the Startup message. Recall, however, that theses file names are in C-series format and must be converted to D-series format using the OLDFILENAME_TO_FILENAME_ procedure before you can pass these names to the FILE_OPEN_ procedure.

```
? INSPECT, SYMBOLS

!Global variables:
STRUCT .CI^STARTUP;                !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULT;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOLUME[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
   END;
   STRING PARAM[0:529];
END;


? NOLIST
? NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS(INITIALIZER)
? LIST

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,MATCH) VARIABLE;

INT .RUCB,
    .START^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN
   !Copy the Startup message into the CI^STARTUP structure:
   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;


PROC INITIAL MAIN;
BEGIN
   CALL INITIALIZER(!rucb!,
                    !passthru!,
                               START^IT);
END;
```

Using ASSIGNs and PARAMs

To use the values provided with ASSIGN or PARAM commands, a program must specifically request to receive the Assign and Param messages. You can use the INITIALIZER procedure to request this information.

If your program requests ASSIGN and PARAM information, then the TACL process sends one Assign message for each assignment currently in effect and one Param message containing all parameter assignments currently in effect.

Note that having received the Assign and Param messages, it is still up to the receiving process to interpret the information.

The Assign message identifies the logical file referred to in the program and the actual file that will be equated to the logical file. Optionally, file characteristics are also provided. Refer to Section 2, Using the File System, Section 3, Coordinating Concurrent File Access, and Section 5, Communicating With Disk Files, for information about the file characteristics.

File names in Assign messages are of C-series internal format. Therefore, you must convert each file name to a valid D-series file name before passing it to a D-series procedure call. If the C-series file name is fully qualified, then you can use the OLDFILENAME_TO_FILENAME_ procedure. If the file name is not fully qualified, you should use the FNAMECOLLAPSE procedure.

The structure of the Assign message follows.

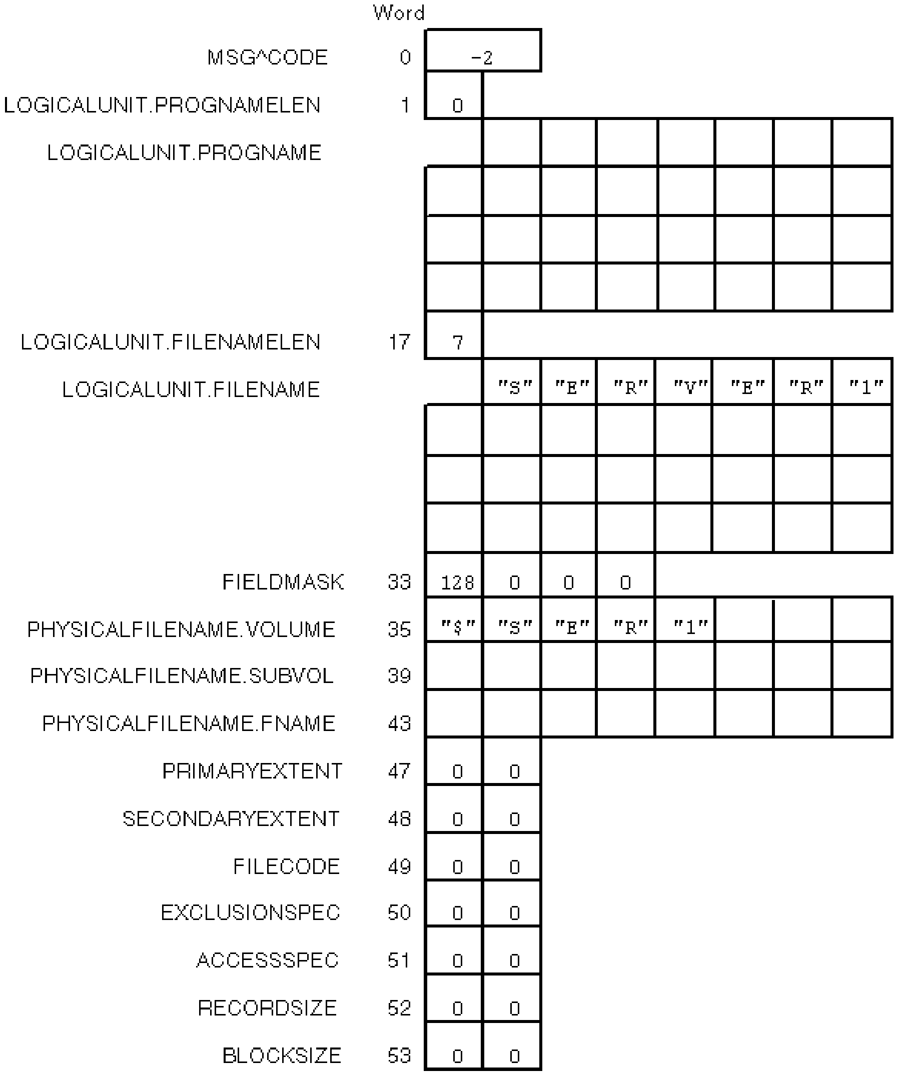
Structure of an Assign message:

```
STRUCT CI^ASSIGN;
BEGIN
   INT MSG^CODE;                !word 0 –  value -2
   STRUCT LOGICALUNIT;
   BEGIN
      STRING PROGNAMELEN;       !word 1 –  program name
length,
                                !0:31 bytes
      STRING PROGNAME[0:30];    !program name (blank-padded)
      STRING FILENAMELEN;       !word 17 – file name length,
                                !0:31 bytes
      STRING FILENAME[0:30];    !file name (blank-padded)
   END;
   INT(32) FIELDMASK;           !word 33 – bit mask to
                                !indicate which of the
                                !following fields were
                                !supplied (1 = supplied)
                                !.<0> = physical file name
                                !.<1> = primary extent size
                                !.<2> = secondary extent size
                                !.<3> = file code
                                !.<4> = exclusion code
                                !.<5> = access specifier
                                !.<6> = record size
                                !.<7> = block size

  STRUCT PHYSICALFILENAME;
   BEGIN
      INT VOLUME[0:3];          !word 35 - physical file name
      INT SUBVOL[0:3];
      INT FNAME[0:3];
   END;
   INT PRIMARYEXTENT;           !word 47 – primary extent
                                !size
   INT SECONDARYEXTENT;         !word 48 – secondary extent
                                !size
   INT FILECODE;                !word 49 – file code
   INT EXCLUSIONSPEC;           !word 50 – exclusion mode:
                                ! %00 if SHARED
                                ! %20 if EXCLUSIVE
                                ! %60 if PROTECTED
   INT ACCESSSPEC;              !word 51 – access mode:
                                ! %0000 if read/write
                                ! %2000 if read only
                                ! %4000 if write only
   INT RECORDSIZE;              !word 52 – record size
   INT BLOCKSIZE;               !word 53 – block size
END;
```

Suppose a program logically refers to a process as SERVER1, then the following ASSIGN command associates this logical name with the actual server name $SER1:

```
1>  ASSIGN SERVER1,$SER1
```

The Assign message received by the program is as follows:


Word
MSG^CODE 0 -2
LOGICALUNIT.PROGNAMELEN 1 0
LOGICALUNIT.PROGNAME
LOGICALUNIT.FILENAMELEN 17 7
LOGICALUNIT.FILENAME "S" "E" "R" "V" "E" "R" "1"
FIELDMASK 33 128 0 0 0
PHYSICALFILENAME.VOLUME 35 "$" "S" "E" "R" "1"
PHYSICALFILENAME.SUBVOL 39
PHYSICALFILENAME.FNAME 43
PRIMARYEXTENT 47 0 0
SECONDARYEXTENT 48 0 0
FILECODE 49 0 0
EXCLUSIONSPEC 50 0 0
ACCESSSPEC 51 0 0
RECORDSIZE 52 0 0
BLOCKSIZE 53 0 0

VST041.VSD

Note. In the figure above, the empty boxes all represent blank characters.

All parameter values assigned using PARAM commands are delivered to the starting process in one Param message:

Structure of a Param message:

```
STRUCT CI^PARAM;
BEGIN
   INT MSG^CODE;                    !word 0 –  value -3
   INT NUMPARAMS;                  !word 1 –  number of
                                    ! parameters in this
message
   STRING PARAMETERS[0:1023];      !word 2 –  the parameters
END;
```

The structure of each parameter in the PARAMETERS field:

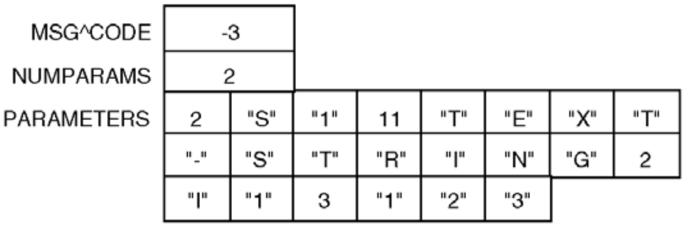
```
param[0]            = "n," length in bytes of parameter name
param[1] FOR n      = parameter name
param[n+1]          = "v," length in bytes of parameter value
param[n+2] FOR v    = parameter value
```

The maximum length of a Param message is 1028 bytes.

Assume that the following PARAM command has been issued at the TACL prompt:

```
PARAM S1 TEXT-STRING
PARAM I1 123
```

The Param message requested by the new process contains the following:



MSG^CODE	-3							
NUMPARAMS	2							
PARAMETERS	2	"S"	"1"	11	"T"	"E"	"X"	"T"
	"-"	"S"	"T"	"R"	"I"	"N"	"G"	2
	"I"	"1"	3	"1"	"2"	"3"		

VST042.VSD

Using INITIALIZER to Read Assign and Param Messages

To request the TACL process to send all Assign and Param messages, your process can again use the INITIALIZER procedure. In addition to reading the Startup message, the INITIALIZER procedure reads the Assign and Param messages if bit 0 of the *`flags`* parameter is set to zero (the default setting). As with the Startup message, you can process the Assign or Param messages by supplying the name of a message-processing procedure as a parameter in the call to INITIALIZER.

The following example reads the Startup message, requests and reads the Assign and Param messages, and calls user-supplied procedures to process each of these messages:

```
CALL INITIALIZER(!rucb!,
                 !passthru!,
                 START^PROC,
                 PARAMS,
                 ASSIGN^NAME);
```

The START^PROC procedure processes the Startup message. The PARAMS procedure processes the Param message, and the ASSIGN^NAME procedure processes Assign messages.

Processing Assign Messages

The following example requests, reads, and processes Assign messages. Here, the process is expecting two assignments: one for the variable SERVER1 and one for the variable SERVER2. A real program will usually also save the Startup message, if only to apply the default values to the file names from the Assign messages.

This example includes two user-written procedures whose combined actions are summarized as follows:

1. The main procedure calls the INITIALIZER procedure, passing it the name of the procedure that will process the Assign messages.
2. The INITIALIZER procedure reads the Startup message and any Assign or Param messages.
3. The INITIALIZER procedure calls the ASSIGN^NAME procedure for each Assign message received.
4. The ASSIGN^NAME procedure checks the Assign message for a logical file name of SERVER1 or SERVER2. If the logical file name is SERVER1, then the actual file name supplied in the message is equated with the logical name SERVER1. If the logical file name is SERVER2, then the actual file name is equated with SERVER2.
5. Once all Assign messages have been processed, the INITIALIZER procedure reads the Param message. Because no procedure is specified for processing the Param message, its contents are ignored.

6. INITIALIZER returns control to the main procedure.

```
? INSPECT, SYMBOLS

!Global variables:
INT SERVER1[0:11];                !Names of two server
INT SERVER2[0:11];                ! processes

? NOLIST
? SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER)
? DATAPAGES = 2
? LIST

PROC ASSIGN^NAME(RUCB,
                 ASSIGN^DATA,
                 MESSAGE,
                 LENGTH,
                 MATCH) VARIABLE;

INT .RUCB,
    .ASSIGN^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN
   STRING .SMESSAGE := @MESSAGE '<<' 1;    !Byte pointer to
                                           ! Assign message

!  If the logical file in the message is "SERVER1," then
!  the SERVER1 file name is set equal to the Tandem file
!  name provided in the message:

   IF  (SMESSAGE[35] = "SERVER1") AND  (SMESSAGE[34] = 7)
          THEN SERVER1 ':=' MESSAGE[35] FOR 12;

!  If the logical file is "SERVER2," then the SERVER2
!  file name is set equal to the Tandem file name provided
!  in the message:

   IF (SMESSAGE[35] = "SERVER2") AND (SMESSAGE[34] = 7) THEN
          SERVER2 ':=' MESSAGE[35] FOR 12;
END;

PROC INITIAL MAIN;
BEGIN
   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    !startupproc!,
                    !paramsproc!,
                    ASSIGN^NAME);
END;
```

Processing the Param Message

This example requests, reads, and processes a Param message. Here, the process expects only one PARAM named P1. Again two user procedures perform the processing as follows:

1. The main procedure calls INITIALIZER and supplies it with the name of the procedure that processes Param messages.
2. The INITIALIZER procedure reads the Startup message and any Assign messages.
3. The INITIALIZER procedure reads the Param message and calls the PARAMS procedure.
4. The PARAMS procedure scans the Param message looking for the parameter name P1. When it finds the P1 parameter, it returns the parameter value to a global variable along with the parameter length.

   Note that if you wanted to retrieve more parameter values from the Param message, you would start the scan again for each additional parameter.

```
? INSPECT, SYMBOLS

!Global variables:
STRING .PARAM1[0:255];      !parameter value
INT PARAM1^LEN;             !parameter length
INT PARAM1^PRESENT;         != 1 if P1 present, 0 if not
                            ! present

? NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS(INITIALIZER)
? LIST

PROC PARAMS(RUCB,
            PARAM^DATA,
            MESSAGE,
            LENGTH,
            MATCH) VARIABLE;

INT .RUCB,
    .PARAM^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN
   INT NUMPAR, I;      !number of parameters in Param message
   STRING .PTR;        !pointer to Param message

!  Get the number of parameters in the message:

   NUMPAR := MESSAGE[1];
```

```
!  Point to the first parameter:

   @PTR := @MESSAGE[2] '<<'1;

!  Set global value to false to indicate no P1 found yet:

   PARAM1^PRESENT := 0;

!  Loop for each parameter until P1 found:

   FOR I := 1 TO NUMPAR DO
   BEGIN

   !  If length and name match then P1 found:

      IF PTR = 2 AND PTR[1] = "P1" THEN
      BEGIN

      !  Advance the pointer to the value for P1:
      !  and save the value for P1 in the global PARAM1:

         @PTR := @PTR '+' PTR '+' 1;
         PARAM1 ':=' PTR[1] FOR PTR;

      !  Save the parameter length:

         PARAM1^LEN := PTR;

      !  Set the PARAM1^PRESENT flag to true:

         PARAM1^PRESENT := 1;
         RETURN;
      END;

   !  Skip the parameter name and parameter value
   !  then loop again to try the next parameter:

      @PTR := @PTR '+' PTR '+' 1;
      @PTR := @PTR '+' PTR '+' 1;
   END;
END;

PROC INITIAL MAIN;
BEGIN
   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    !startupproc!,
                    PARAMS);
    .
    .
END;
```

Setting a Timeout Value for INITIALIZER

Normally, INITIALIZER waits 60 seconds for a startup sequence message to arrive on $RECEIVE. If no message is received in that time, INITIALIZER times out, assuming that the sending process has either terminated or is not going to send a startup sequence. For most purposes, the 60-second default value is appropriate; however, depending on expected system load, you might prefer to set the timeout interval to some other value.

You use the *timeout* parameter to the INITIALIZER procedure to set the timeout value. The supplied value is in units of 0.01 second. Choose the value carefully: small values time out often, and large values cause INITIALIZER to wait for long periods of time in cases where the sending process has terminated or does not send the appropriate startup messages.

The following example sets the timeout interval to 120 seconds:

```
INT(32)   TIMELIMIT;
 .
 .
TIMELIMIT := 12000D;
CALL INITIALIZER(!rucb!,
                 !passthru!,
                 !startupproc!,
                 !paramproc!,
                 !assignproc!,
                 !flags!,
                 TIMELIMIT);
```

Reading the Startup Sequence Without INITIALIZER

To read the standard startup sequence of messages without using the INITIALIZER procedure, your program must do the following:

1. Open $RECEIVE with a receive depth of 1 or more, to allow replies to be made to the startup messages.
2. Read the Open message from $RECEIVE and reply to it with a reply code of 0.
3. Read the Startup message and process it (for example, by copying it into a global area). Once you have processed the Startup message, you must take one of the following actions if you want to receive the remaining messages (Assign and Param messages) of the startup sequence:
    - Reply to the Startup message with a reply code of 70, which instructs the TACL process to continue sending the startup sequence of messages.
    - Reply to the Startup message with a reply code of 0, but with a reply length of four bytes. To receive Assign messages, bit 0 of the first byte must be 1.

To receive Param messages, bit 1 of the first byte must be 1. All other bits must be 0.

If you do not reply to the Startup message or if you reply with some other reply code, then your process will receive no further startup sequence messages.

4. Read and process each Assign message if there are any and reply to each Assign message with a reply code of 0.
5. Read the Param message if there is one, process it, and reply to it with a reply code of 0.
6. If unexpected messages are received, the program must reply to them with the reply code 100 and continue.
7. Read the Close message and close $RECEIVE.

The following sample procedure performs the above tasks using reply code 70 to request Assign and Param messages.

```
PROC READ^STARTUP^SEQUENCE;
BEGIN
   STRING .RCV^NAME[0:ZSYS^VAL^LEN^FILENAME - 1];
   INT     RCV^NUM,
           NAMELEN,
          .RCV^BUF[0:514],
           COUNT^READ,
           REPLY^CODE = 0,
           ERROR,
           S^PTR;
   LITERAL RCV^DEPTH = 1,
           RCV^COUNT = 1030,
           CLOSE^MSG = -104;

!  Open $RECEIVE:

   RCV^NAME ':=' "$RECEIVE" -> @S^PTR;
   NAMELEN := @S^PTR '-' @RCV^NAME;
   ERROR := FILE_OPEN_(RCV^NAME:NAMELEN,
                       RCV^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       RCV^DEPTH);
   IF ERROR <> 0 THEN ...;

!  Read the Open message from $RECEIVE:

   CALL READUPDATE(RCV^NUM,RCV^BUF,RCV^COUNT,COUNT^READ);
   CALL FILE_GETINFO_(RCV^NUM,ERROR);
   IF ERROR <> 6 THEN CALL PROCESS_STOP_;

!  Loop until Close message received

   WHILE RCV^BUF <> CLOSE^MSG DO
   BEGIN
```

```
            CASE RCV^BUF OF
            BEGIN
               -1 ->  BEGIN
               !  Process Startup message
                   .
                   .
                  REPLY^CODE := 70;
               END;

               -2 -> BEGIN
               !  Process Assign message
                   .
                   .
                  REPLY^CODE := 0;
               END;

               -3 -> BEGIN
               !  Process Param message
                   .
                   .
                  REPLY^CODE := 0;
               END;
               !  Process illegal messages
               OTHERWISE->REPLY^CODE := 100;
            END;

         !  Reply to last message received:

            CALL REPLY(!buffer!,
                       !write^count!,
                       !count^written!,
                       !message^tag!,
                       REPLY^CODE);

         !  Read next message from $RECEIVE:

            CALL READUPDATE(RCV^NUM,RCV^BUF,RCV^COUNT,COUNT^READ);
            CALL FILE_GETINFO_(RCV^NUM,ERROR);
            IF ERROR <> 6 THEN CALL PROCESS_STOP_;
         END;

      !  Reply to Close message:

         CALL REPLY(!buffer!,
                    !write^count!,
                    !count^written!,
                    !message^tag!,
                    0);

      !  Close $RECEIVE:

         ERROR := FILE_CLOSE_(RCV^FNUM);
         IF ERROR <> 0 THEN ...
      END;
```

Waking the TACL Process

The Wakeup message can be sent by an application process to the TACL process to cause the TACL process to return to command-input mode. In command-input mode, TACL prompts for commands.

You might want to use the Wakeup message for one of the following reasons:

- If a process takes BREAK ownership away from the TACL process and then becomes unresponsive, pressing BREAK does not return control to the TACL process because it no longer owns BREAK. In addition, the program that does own BREAK is not checking for BREAK. In this case, you can run a program from another terminal to send the TACL process the Wakeup message. The TACL prompt then appears on the user’s terminal, allowing the user to continue.

  See Section 10, Communicating With Terminals, for a discussion of the BREAK key.

- Your program might initially interact with the user, then start some long processing without user interaction. Before starting the long processing, your process could send the TACL process a Wakeup message to allow the user to continue.

The Wakeup message is made up of one word containing a message code of -20:

Structure of the Wakeup message:

```
STRUCT WAKEUP^MSG;
BEGIN
   INT MSGCODE;            !value -20
END;
```

To send the message code to the TACL process, you first need to open the TACL process as you would any process. Then you write the Wakeup message to the returned file number. The following example shows how:

```
?INSPECT, SYMBOLS
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS(FILE_OPEN_,WRITE,
?                              DEBUG)
?LIST

PROC AWAKE^CI MAIN;
BEGIN

   LITERAL MAXLEN = 256;
   STRUCT WAKEUP^MSG;
   BEGIN
      INT MSGCODE;
   END;


   INT LENGTH;                        !length of CI name
   INT CI^NUM;                        !number of open CI file
   STRING .CI^NAME[0:MAXLEN - 1];   !CI process name

!  Set up the correct message code:

   WAKEUP^MSG.MSGCODE := -20;

!  Open the TACL process:

   CI^NAME ':=' "$G55";
   LENGTH := 4;
   ERROR := FILE_OPEN_(CI^NAME:LENGTH,CI^NUM);
   IF ERROR <> 0 THEN CALL DEBUG;

!  Write the Wakeup message to the TACL process:

   CALL WRITE(CI^NUM,WAKEUP^MSG,2);
   IF <> THEN CALL DEBUG;
END;
```

Note. The recommended way of determining the name of your creator process is to obtain its process handle using the PROCESSPAIR_GETINFO_ procedure, then pass the process handle to the PROCESSHANDLE_DECOMPOSE_ procedure to obtain the process name. For brevity, however, the above example simply uses the hard-coded name. For information about the PROCESSPAIR_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures, see Section 16, Creating and Managing Processes, or the *Guardian Procedure Calls Reference Manual.*

Causing the TACL Process to Display Text

You can cause a TACL process to display text on the terminal by sending a Display message to it. The TACL process displays the text sent in the message before the next command prompt.

The message code for the Display message is -21. The complete structure of the message is shown below:

Display message structure:

```
STRUCT DISPLAY^MSG;
BEGIN
   INT MSGCODE;            !value -21
   STRING TEXT[0:n-1];     !n <= 132
END;
```

To make the TACL process display text, you must first open the TACL process as you would any process, then write the Display message to the open file number. The following example shows how:

```
?INSPECT, SYMBOLS
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS(FILE_OPEN_,WRITE,
?                              DEBUG)
?LIST

PROC EGCI MAIN;
BEGIN

   LITERAL MAXLEN = 256;
   STRUCT .DISPLAY^MSG;
   BEGIN
      INT MSGCODE;                    !set to -31
      STRING TEXT[0:131];             !place holder for text
   END;

   INT LENGTH;                        !length of process file
                                      ! name
   INT CI^NUM;                        !file number of CI
   STRING .CI^NAME[0:MAXLEN - 1];     !file name of CI

!  Fill in the display message with -21 for the message
!  code and the text to be displayed:

   DISPLAY^MSG.MSGCODE := -21;
   DISPLAY^MSG.TEXT ':=' "Display this Text";


!  Open the TACL process:

   CI^NAME ':=' "$G55";
   LENGTH := 4;
   ERROR := FILE_OPEN_(CI^NAME:LENGTH,CI^NUM);
   IF ERROR <> 0 THEN CALL DEBUG;

!  Write the display message to the TACL process:

   CALL WRITE(CI^NUM,DISPLAY^MSG,19);
   IF <> THEN CALL DEBUG;
END;
```

Note. The recommended way of determining the name of your creator process is to obtain its process handle using the PROCESSPAIR_GETINFO_ procedure, then pass the process handle to the PROCESSHANDLE_DECOMPOSE_ procedure to obtain the process name. For brevity, however, the above example simply uses the hard-coded name. For information about the PROCESSPAIR_GETINFO_ and PROCESSHANDLE_DECOMPOSE_ procedures, see Section 16, Creating and Managing Processes, or the *Guardian Procedure Calls Reference Manual.*

9 Communicating With Devices

In addition to introducing some of the major features of the I/O subsystem, this section discusses mechanisms that your program can use to communicate with devices in general. These include:

- Device access through device names and logical device numbers
- Control through SETMODE, CONTROL, and SETPARAM procedure calls
- Access to device-specific status information such as device type

The information presented here provides an introduction to subsequent sections that discuss communication with specific device types: Section 10, Communicating With Terminals, provides details on communicating with terminals; Section 11, Communicating With Printers, discusses access to printers; and Section 12, Communicating With Magnetic Tape, provides information about communicating with magnetic tape. Refer to the data communications manuals for information regarding communication with other types of communications lines.

Overview of I/O Subsystem

Before this section discusses how to access and control devices, it is necessary for you to understand some basic features of the I/O subsystem.

When your application program accesses a device, it does so with a call to the file system. The file system sends a message to the I/O process that processes all requests for the specified device.

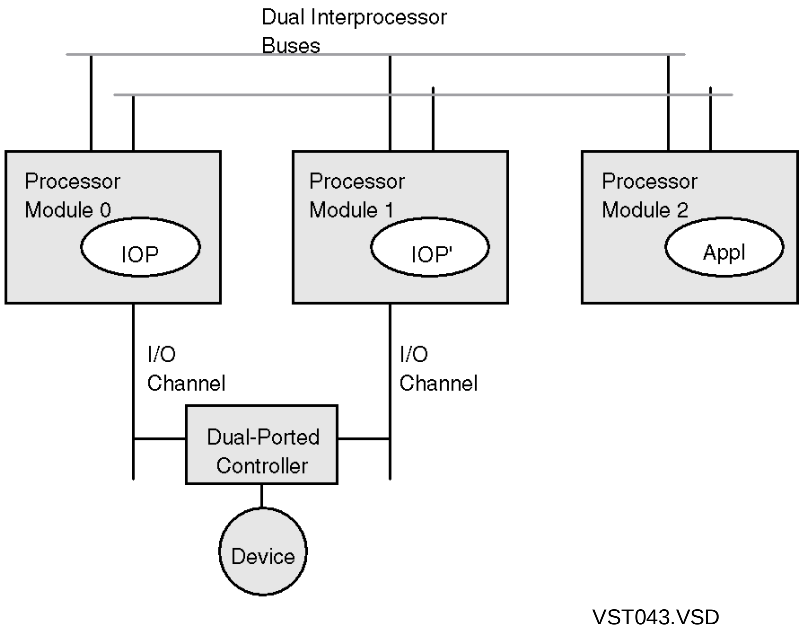
Usually, two independent paths exist to each device to ensure that the device is always accessible. Therefore, if any failure occurs within the system, at least one path to the device is still operable.

Hardware and software work together to provide device fault tolerance in a way that is invisible to the application process. Figure 9-1 shows the architectural components of the system that support device fault tolerance.

The I/O process (IOP) contains the code that actually performs the operation on the device. Note that two IOPs usually exist for the device: a primary IOP and a backup IOP. Although only the primary IOP is active at any time, the backup is ready to become the primary if for any reason the primary is unable to continue processing.

For devices that are configured for continuous availability, redundant hardware ensures that a hardware path is always available to the device. Checkpointing between the primary IOP and backup IOP makes sure that a backup IOP is always ready to become the primary should the need arise.

Figure 9-1. Overview of the I/O Subsystem


Dual Interprocessor Buses
Processor Module 0
IOP
Processor Module 1
IOP'
Processor Module 2
Appl
I/O Channel
I/O Channel
Dual-Ported Controller
Device
VST043.VSD

Addressing Devices

At system-generation time, each device is given a device name and a device number. Refer to the appropriate system generation manual for details on how these names and numbers are assigned.

Recall from Section 2, Using the File System, that you can use either the device name or the logical device number to identify a device:

```
[node-name.]{device-name|ldev-number}
```

The device name can be up to 8 alphanumeric characters long: the first character must be a dollar sign ($), and the second must be alphabetic. The logical device number is a decimal number in the range 5 through 34492, and it is always preceded by a dollar sign. (Logical device numbers 0 through 4 are reserved for operating-system processes.)

So how do you know the name of the device you want your program to communicate with? It depends on the device in question. If you simply want to communicate with the home terminal of the process, you can get the home terminal name using the PROCESS_GETINFO_ procedure:

```
CALL PROCESS_GETINFO_(!process^handle!,
                      !file^name:maxlen!,
                      !file^name^len!,
                      !priority!,
                      !moms^processhandle!,
                      TERMINAL^NAME:MAXLEN,
                      LENGTH);
```

To communicate with the IN or OUT file specified in the Startup message, get the file name from the Startup message using the INITIALIZER procedure as described in Section 8, Communicating With a TACL Process.

If you need to communicate with a specific terminal, you can find the name of the terminal by issuing the WHO command on the terminal in question:

```
9> WHO
Home terminal: \SYS.$MASTER
 .
 .
```

For terminals you cannot physically access, or for printers, magnetic tape units, or any other device, you should ask your system administrator for information about the device names or logical device numbers configured on your system.

Accessing Devices

Device access is the same as for any file. However, you cannot create device files in an application program. The files are created during system configuration along with the device names and logical device numbers.

The basic actions of opening, closing, reading from, and writing to devices have already been discussed in Section 2, Using the File System. In addition to these basic operations, however, your program can also retrieve information about a specific device; see Getting Device Information later in this section.

Controlling Devices

You use the following procedures to control devices:

- CONTROL performs device-specific I/O operations like sending a form feed to a printer or terminal or rewinding a magnetic tape.
- CONTROLBUF is similar to CONTROL in that it specifies operations to be performed on a device. CONTROLBUF, however, also enables buffered data to be passed to a device. For example, you can use CONTROLBUF to load the direct access vertical format unit (DAVFU) of a subtype 4 printer.
- SETMODE provides several options for setting the operational mode of a device; for example, the security level for future disk accesses, parity checking on a terminal, the form length for a printer, or the packet size for an X.25 communications line.
- SETMODENOWAIT provides the same functions as SETMODE but returns to the caller immediately. SETMODENOWAIT finishes with a call to AWAITIO in the same way as any other nowait operation. Refer to Section 4, Using Nowait Input/Output, for details of nowait I/O.
- SETPARAM is concerned with setting and getting parameters for controlling data communications lines, and for establishing BREAK ownership for a terminal.

For complete details of command syntax for the above procedures, see the *Guardian Procedure Calls Reference Manual*. This guide primarily describes CONTROL and SETMODE (or SETMODENOWAIT). For programming information on the SETPARAM procedure, refer to the data communications manuals.

SETMODE usually sets a condition within which a device will operate; the condition remains in effect as long as the file remains open or until the condition is changed with another call to SETMODE. CONTROL performs an operation on a device. For example, you use the SETMODE procedure to set the form length for a printer; SETMODE sets the condition. But to actually issue a form-feed character, you use the CONTROL procedure.

The next three sections of this manual describe how SETMODE and CONTROL are used to control terminals, printers, and magnetic tape drives. Specific examples are given there, since use of these procedures is specific to the device type.

For information on the use of SETMODE and CONTROL with other data communications lines, refer to the data communications manuals.

Getting Device Information

On D-series systems, you can obtain detailed information about devices either by supplying a logical device number to the DEVICE_GETINFOBYLDEV_ procedure or by supplying a device name to the DEVICE_GETINFOBYNAME_ procedure. DEVICE_GETINFOBYLDEV_ also has the option of searching for the next logical device number above the one specified.

On G-series systems, you can use CONFIG_GETINFO_BYNAME2_ and CONFIG_GETINFO_BYLDEV2_ procedures, respectively.

The information returned by each pair of procedures is similar. In either case, you can obtain logical device information or physical device information about the primary and backup I/O processes.

Logical information returned by the device-information procedures includes (but is not limited to) the following:

- The logical device number of the device for which information is being returned. This information is most useful when using the search option of the DEVICE_GETINFOBYLDEV_ procedure.
- The CPU number and process identification number (PIN) for the primary and backup I/O processes that control access to the specified device.
- The device type and subtype.

For a complete list of logical information, see the *Guardian Procedure Calls Reference Manual*.

Physical information returned for the primary and backup I/O processes includes (but is not limited to):

- A file-system error number returned by the I/O process.
- Subtype information about both halves of a mirrored disk.
- Path information about each potential path to the I/O device, indicating whether the path is currently configured, whether it is currently in use by the I/O process, and its operational status.

For a complete list of physical information, see the *Guardian Procedure Calls Reference Manual*.

The following example uses the DEVICE_GETINFOBYLDEV_ procedure to return logical and physical information about all configured devices. It does this by searching on a logical device number as indicated earlier. To do this, you set bit 15 of the *`options`* parameter to 1.

```
!Data structure for returned logical device information:
STRUCT .LOGICAL^INFO;
BEGIN
   INT(32) LDEV;         !logical device number
   INT     PRIMARY^CPU;  !CPU where primary I/O process runs
   INT     PRIMARY^PIN;  !PIN of primary I/O process
   INT     BACKUP^CPU;   !CPU where backup I/O process runs
   INT     BACKUP^PIN;   !PIN of backup I/O process
   INT     TYPE;         !device type
   INT     SUBTYPE;      !device subtype
   INT     RECORD^SIZE;  !record length in bytes
   INT     FLAGS;        !<1> set if device is TMF audited
                         !<2> set if device dynamically
                         !    configured
                         !<3> set if logically demountable
                         !<4> set if device has subdevices
                         !    that can be opened
END;

!Template structure for physical device information:
STRUCT PHYSICAL^INFO (*);
BEGIN
   INT STATUS;          !file-system error number from IOP
   INT PRIMARY^SUBTYPE; !device subtype of primary half of
                        ! mirrored disk
   INT MIRROR^SUBTYPE;  !device subtype of mirrored half of
                        ! mirrored disk
   INT FLAGS;           !<1> set if device has physical
                        !    devices
                        !<2> set if current primary IOP
   STRUCT PATH0;        !substructure for path 0
   BEGIN
      INT FLAGS;        !<1> set if path configured
                        !<2> set if path currently in use by
                        !    IOP
      INT CHANNEL;      !channel number of path
      INT CONTROLLER;   !controller number of path
      INT UNIT;         !unit number of path
      INT STATE;        !operational status of path
   END;
   STRUCT PATH1;        !substructure for path 1
   BEGIN
      INT FLAGS;
      INT CHANNEL;
      INT CONTROLLER;
      INT UNIT;
      INT STATE;
   END;
   STRUCT PATH2;        !substructure for path 2
   BEGIN
      INT FLAGS;
      INT CHANNEL;
      INT CONTROLLER;
      INT UNIT;
      INT STATE;
   END;
   STRUCT PATH3;        !substructure for path 3
   BEGIN
      INT FLAGS;
      INT CHANNEL;
      INT CONTROLLER;
      INT UNIT;
      INT STATE;
   END;
END;

!Variables for DEVICE_GETINFOBYLDEV_ procedure:
INT ERROR,
    LOGICAL^DEVICE,  !logical device number
    .L^INFO[0:9],    !logical device information
    L^INFO^MAXLEN,   !length of output buffer
    L^INFO^LEN,      !length in bytes of returned data
    .P^INFO[0:23],   !primary IOP information
    P^INFO^MAXLEN,   !length of output buffer
    P^INFO^LEN,      !length in bytes of returned data
    .B^INFO[0:23],   !backup IOP information
    B^INFO^MAXLEN,   !length of output buffer
    B^INFO^LEN,      !length in bytes of returned data
    OPTIONS;         !options parameter



!Data structure for returned physical device information
!related to primary IOP:
STRUCT .PHYSICAL^PRIMARY^INFO (PHYSICAL^INFO);

!Data structure for returned physical device information
!related to backup IOP:
STRUCT .PHYSICAL^BACKUP^INFO (PHYSICAL^INFO);

  .
  .

!Set the maximum lengths of the logical, primary, and backup
!buffers for returned information
L^INFO^MAXLEN := $LEN(LOGICAL^INFO);
P^INFO^MAXLEN := $LEN(PHYSICAL^PRIMARY^INFO);
B^INFO^MAXLEN := $LEN(PHYSICAL^BACKUP^INFO);
```

```
!Set the search option, starting from logical device
!number 0:
OPTIONS.<15> := 1;
LOGICAL^DEVICE := 0;

!Loop until no more logical device numbers:
WHILE ERROR <> 4 DO
BEGIN

   !Get the device information:
   ERROR := DEVICE_GETINFOBYLDEV(
                  LOGICAL^DEVICE;
                  L^INFO,L^INFO^MAXLEN,L^INFO^LEN,
                  P^INFO,P^INFO^MAXLEN,P^INFO^LEN,
                  B^INFO,B^INFO^MAXLEN,B^INFO^LEN,
                  !timeout!,
                  OPTIONS);

   !Copy device information into prepared data structures:
   LOGICAL^INFO ':=' L^INFO FOR (L^INFO^LEN/2);
   PHYSICAL^PRIMARY^INFO ':=' P^INFO FOR (P^INFO^LEN/2);
   PHYSICAL^BACKUP^INFO ':=' B^INFO FOR (B^INFO^LEN/2);
    .
    .
END;
 .
 .
```

Additional Device Information (G-series Only)

Some G-series processes support a new interface for obtaining more device information than was available through the D-series DEVICE_GETINFOBYLDEV_ and DEVICE_GETINFOBYNAME_ procedures. This interface consists of a client calling the CONFIG_GETINFO_BYNAME2_ and CONFIG_GETINFO_BYLDEV2_ procedures, and the device returns information to a system message in response to the call.

In addition to providing information that is common to all types of devices (such as device type and subtype), the new interface allows devices to optionally define and return their own device-dependent information.

The following C program shows how the CONFIG_GETINFO_BYNAME2_ procedure can be used to obtain device information for an arbitrary process, in this case "$EXMPL." This program assumes that the device $EXMPL provides a type for its device-specific information in "example.h" called example_specific_info_type. Other devices either return no device-specific information, or return device-specific information in formats that the programmer specifies.

```
#include <cextdecs>
#include <stdio.h>
#include <string.h>
#include "zsysc"
#include “example.h”
int main()

{

   char  device_name[] = "$EXMPL";
   zsys_ddl_config_getinfo2_def common_info;   /* device-independent info */
   example_specific_info_type   specific_info; /* device-dependent info  */
   short common_len, specific_len;             /* returned sizes of device *
                                                * information            */
   long  error, error_detail;
   error = CONFIG_GETINFO_BYNAME2_

      (device_name             /* Name of device to query */
      ,strlen(device_name)     /* .. and it's length  */
      ,(short *)&common_info   /* Buffer to hold device-independent info */
      ,sizeof(common_info)     /* .. and it's size        */
      ,&common_len             /* .. actual size returned by device */
      ,(char *)&specific_info  /* Buffer to hold $EXMPL-specific info  */
      ,sizeof(specific_info)   /* .. and it's size                      */
      ,&specific_len           /* .. actual size returned by device      */
      ,2000                    /* Wait at most 20 seconds before timeout */
      ,&error_detail           /* More error information                 */
      );

    if(error == 1)
        printf("Filesystem error %d returned\n",error_detail);
    else if(error == 2)
        printf("Parameter %d was invalid\n",error_detail);
    else if(error == 3)
        printf("Parameter %d failed bounds check\n",error_detail);
    else if(error == 4 && error_detail == -1)
        printf("Device returned invalid data to getinfo request\n");
    else if(error == 4)
        printf("Device returned error %d to getinfo request\n",error_detail);
    else {
        /* No error, common_info and specific info now contain valid data
         * of lengths common_len and specific_len */
        . . . .
    }
```

To support the interface for CONFIG_GETINFO_BYNAME2_ procedure or CONFIG_GETINFO_BYLDEV2_ procedure, a device must be coded to handle a new system message from $RECEIVE (-147: ZSYS_VAL_SMSG_CONFIGINFO). The format of this system message is defined in zsysc by the zsys_ddl_smsg_configinfo2_def type. The reply from a device that supports these procedures is expected to have a zsys_ddl_smsg_confinf_reply_def format.

The following code fragment demonstrates how a subsystem might be enhanced to support the CONFIG_GETINFO calls in its $RECEIVE message-handling code. Effectively, the code simply receives a message from $RECEIVE, and branches to the code to handle specific types of system messages. To handle the ZSYS_VAL_SMSG_CONFIGINFO request, the fields of a zsys_ddl_smsg_confinf_reply_def are filled with appropriate values for the device, and REPLYX is called to reply to the request. For more information on processing system messages, see Processing System Messages and Section 9, Communicating With Devices.

```
{

   short replyerr;
   zsys_ddl_smsg_configinfo2_def *ci2req;
   zsys_ddl_smsg_confinf_reply_def   cireply;
   _cc_status cc;

  /* ... code to receive system messages */
  cc = READUPDATEX( recv_fnum, (char *)message, (short)sizeof(message),&len);
  /* ... */

  switch(message[0]) {
  case ZSYS_VAL . . . :
  /* ... other system message types */

  case ZSYS_VAL . . . :

  /* ... other system message types */

  case ZSYS_VAL_SMSG_CONFIGINFO:  /* CONFIG_GETINFO_ request */
      ci2req = (zsys_ddl_smsg_configinfo2_def *)message;
      if(ci2req->z_msgversion != ZSYS_VAL_SMSG_CONFIGINFO_VERS2){
          cc = REPLYX(,,,,FEBADOP);
            /* . . . */

          }

        else {
            cireply.z_msgnumber = ci2req->z_msgnumber;
            cireply.z_msgversion = ci2req->z_msgversion;
            cireply.z_device_type =             /* --<-- */ ;
            cireply.z_device_subtype =          /* --<-- */ ;
            cireply.z_device_record_size =      /* --<-- */ ;
            cireply.z_logical_status = 0        /* --<-- */ ;
            strcpy(cireply.z_config_name, "CONFIG_EXAMPLE_PROC");
            cireply.z_config_name_len = strlen(cireply.z_config_name);
            strcpy(cireply.z_subsys_manager = "$MANAGR";
            cireply.z_subsys_manager_len = strlen(cireply.z_subsys_manager);
            PROCESSHANDLE_GETMINE_((short *)&cireply.z_primary_phandle);
            PROCESSHANDLE_NULLIT_((short *)&cireply.z_backup_phandle);
            cireply.z_specific_info_len = 0;
            cc = REPLYX( (char *)&cireply /* buffer        */
                       , sizeof(cireply)  /* write_count   */
                       ,                  /* count_written */
                       ,                  /* message tag   */
                       , FEOK
```

```
                    );
                if(_status_ne(cc)) {
                    short err;
                    FILE_GETINFO_(g_recv_fnum, &err);
                    printf("Warning! Error %d on reply to configinfo
                                        sysmsg\n",err);
                }
                return;
            }
            break;
        }
    }
```

10

Communicating With Terminals

This section describes how an application process can communicate with a terminal using file-system procedure calls. The file system can communicate with any terminal whose characteristics can be defined to the system through one of the programs that can configure devices, such as the system-generation program, SYSGEN. Refer to the *System Generation Manual for Terminals and Printers*.

Specifically, this section describes the following topics:

- How to access a terminal: how to open a terminal and how to perform I/O with an open terminal
- How to communicate with a terminal in conversational mode
- How to communicate with a terminal in page mode
- How to manage the BREAK key
- How to recover from errors

For a complete discussion of the programmatic interface to a terminal, refer to the appropriate terminal manual; for example, the *653X Multi-Page Terminal Programmer's Guide*.

Accessing a Terminal

This subsection discusses how to perform basic I/O operations with a terminal as well as how to control general terminal characteristics. The following topics are discussed:

- How to open a terminal for access
- How to transfer data between a computer system and a terminal
- How to time out a user response
- How to control text echo to the terminal
- How to set the mode of data transfer: conversational or page mode
- How to terminate terminal access

The use of the BREAK key is not discussed here. The BREAK key is described later in this section in Managing the BREAK Key.

You access a terminal the same way as you would any file, by using procedure calls to the file system. You use the following procedure calls to perform the indicated tasks with terminals:

AWAITIO[X]	Waits for completion of outstanding I/O operations that are pending on the open terminal.
CANCEL	Cancels the oldest outstanding operation on an open terminal.
CANCELREQ	Cancels a specified operation on an open terminal.
CONTROL	Controls forms movement and modem connection and disconnection. Table 10-1 provides a summary.
FILE_GETINFOBYNAME_	Provides the device type and configured record length of the device specified by name.
FILE_CLOSE_	Stops access to an open terminal.
FILE_GETINFO_	Provides error information and characteristics about an open terminal.
FILE_OPEN_	Establishes communication with a terminal.
PROCESS_GETINFO_	Returns the name of the home terminal of the process.
READ[X]	Waits for and receives information typed at an open terminal.
SETMODE	Sets and clears terminal-related functions. Table 10-2 provides a summary.
SETMODENOWAIT	Acts the same as SETMODE, but the terminal functions are applied in a nowait manner.
SETPARAM	Establishes BREAK key ownership.
WRITE[X]	Sends information to an open terminal.
WRITEREAD[X]	Writes to an open terminal, then waits for data to be read from the same terminal.

Table 10-1 summarizes all CONTROL operations that affect terminal operation.

Table 10-1. Terminal CONTROL Operations

CONTROL Number	Operation
1	Provides forms control
11	Specifies a wait for a modem connection
12	Disconnects a modem

On return from one of the calls listed in Table 10-1, the condition code should be CCE if the CONTROL operation was successful. A condition code of CCL indicates an error.

Table 10-2 summarizes all SETMODE functions that relate to terminal operation.

Table 10-2. Summary of Terminal SETMODE Functions

SETMODE Number	Function
6	Sets system spacing control
7	Sets system auto line feed after receipt of line-termination character
8	Sets the system transfer mode (page mode or conversational mode)
9	Sets the interrupt characters
10	Sets parity checking
11	Sets BREAK ownership (SETPARAM function 3 also sets BREAK mode)
12	Sets the terminal access mode for normal mode or BREAK mode
13	Sets system read termination following receipt of ETX character
14	Sets system read termination following receipt of an interrupt character
20	Sets echo mode
22	Sets the terminal baud rate
23	Sets the number of bits that specify one character
24	Sets parity generation
27	Sets system spacing mode
28	Resets configured values
38	Sets special line-termination mode and the new line-termination interrupt character
67	Enables or disables AUTODCONNECT for full-duplex modems
110	Enables or disables shift in, shift out mode
113	Sets the screen size

On return from a call to SETMODE for one of the calls listed in Table 10-2, the condition code should be CCE if the function was performed successfully. A condition code of CCL indicates an error. A condition code of CCG indicates that the attempted SETMODE function is invalid for the type of device.

Once you change the mode of terminal using one of these SETMODE functions, the change remains in effect for all processes that use the terminal until you revoke the change by issuing another call to the SETMODE procedure.

You can use the *`last-params`* parameter of the SETMODE procedure to obtain the previous settings associated with a function.

For complete details on any of the procedures, CONTROL operations, and SETMODE functions listed here, refer to the *Guardian Procedure Calls Reference Manual*.

Opening a Terminal

You establish communication with a terminal the same way as you would for any file, by issuing a call to the FILE_OPEN_ procedure. You must supply the FILE_OPEN_ procedure with the terminal name. FILE_OPEN_ returns a file number that you use to make subsequent access to the terminal.

The most common way of opening a terminal is by opening the IN and OUT files whose names are supplied in the Startup message. A less useful approach is to open the home terminal of the process. These operations are described below. Note, however, that opening terminals is not limited in this way; you can open any terminal that you know the name or logical device number of.

Multiple concurrent opens for a terminal are permitted but limited depending on which driver controls the terminal.

Opening the IN and OUT Files

Usually, the only terminals that an application needs to open are those terminals named in the Startup message as the IN file and the OUT file. For example, if you have saved your Startup message in a structure called CI^STARTUP as described in Section 8, Communicating With a TACL Process, you would open your IN and OUT files as follows. Note that the Startup message provides 12-word file names compatible with C-series systems. To convert these file names into D-series file names, you can use the OLDFILENAME_TO_FILENAME_ procedure.

```
LITERAL MAXLEN = 256;
STRING IN^TERMNAME[0:MAXLEN - 1];
STRING OUT^TERMNAME[0:MAXLEN - 1];
INT INFILE^LENGTH, OUTFILE^LENGTH;

STRUCT .CI^STARTUP;                !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULT;
   BEGIN
```

```
      INT VOLUME[0:3];
      INT SUBVOLUME[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
   END;
   STRING PARAM[0:529];
END;

!  Convert the C-series file name into a D-series file name:

   ERROR := OLDFILENAME_TO_FILENAME_(CI^STARTUP.INFILE,
                                      IN^TERMNAME:MAXLEN,
                                      INFILE^LENGTH);

!  Open the IN file:

   ERROR := FILE_OPEN_(IN^TERMNAME:INFILE^LENGTH,
                       IN^TERMNUM);
   IF ERROR <> 0 THEN ...

!  Convert the C-series file name into a D-series file name:

   ERROR := OLDFILENAME_TO_FILENAME_(CI^STARTUP.OUTFILE,
                                     OUT^TERMNAME:MAXLEN,
                                     OUTFILE^LENGTH);

!  Open the OUT file:

   ERROR := FILE_OPEN_(OUT^TERMNAME:OUTFILE^LENGTH,
                       OUT^TERMNUM);
   IF ERROR <> 0 THEN ...
```

Note. The IN file and OUT file names are not necessarily terminal names. They can be any valid file name as defined for C-series systems.

Opening the Home Terminal

To open the home terminal (the terminal that starts the application), you can find out the name of the terminal using the PROCESS_GETINFO_ procedure. The PROCESS_GETINFO_ procedure returns the terminal name and the name length. You pass both of these parameters to the FILE_OPEN_ procedure.

```
ERROR :=  PROCESS_GETINFO_(!process^handle!,
                           !file^name:maxlen!,
                           !file^name^len!,
                           !priority!,
                           !moms^processhandle!,
                           TERMINAL^NAME:MAXLEN,
                           LENGTH);
IF ERROR <> 0 THEN ...
```

```
ERROR := FILE_OPEN_(TERMINAL^NAME:LENGTH,
                    TERMNUM);
IF ERROR <> 0 THEN ...
```

Transferring Data Between Application and Terminal

Use the WRITE[X], READ[X], and WRITEREAD[X] procedures to transfer information between the application program and the terminal. The following paragraphs describe how to do this.

Writing to a Terminal

Use the WRITE[X] procedure to write to a terminal as you would any file:

```
SBUFFER ':=' "Some text to display on the terminal "
             -> @S^PTR;
CALL WRITEX(TERMNUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

The above example writes the text placed into the string buffer to the terminal. The WRITEX procedure usually appends a carriage return/line feed sequence to the specified bytes.

Reading From a Terminal

Use the READ[X] procedure to read from a terminal:

```
CALL READX(TERMNUM,
           SBUFFER,
           BUFSIZE,
           COUNT^READ);
IF <> THEN ...
```

Here, the application process reads up to BUFSIZE (the size of SBUFFER) characters into the buffer and returns the number actually read in the variable COUNT^READ. On issuing the READX procedure call, the process waits for the read operation to finish, which is an indefinite time. The read operation finishes when one of the following conditions is satisfied:

- The user issues the line-termination or page-termination character (usually associated with the RETURN key).
- The number of characters specified in BUFSIZE have been read.

Pressing CONTROL/Y or pressing the BREAK key also has the effect of terminating the input.

Writing to and Reading From a Terminal in One Operation

Use the WRITEREADX procedure to ensure that the system is ready to receive data from the terminal immediately after you write to the terminal. This feature is useful when conversationally prompting a terminal user for input. The WRITEREADX procedure combines a write operation and a read operation in one procedure call.

The following example prompts the terminal user with a colon and then waits for the reply. Note that the same buffer is used for the prompt as for the reply.

```
SBUFFER ':=' ": ";
WCOUNT := 1;
CALL WRITEREADX(TERMNUM,
                SBUFFER,
                WCOUNT,
                BUFSIZE,
                COUNT^READ);
```

```
IF <> THEN ...
```

This example writes one character (the colon) from the buffer to the terminal. The application then waits for a response from the terminal. The response in this case is either BUFSIZE characters of data or fewer than BUFSIZE characters terminated by the user pressing the line-termination or page-termination key (usually the carriage return).

Note that WRITEREADX does not issue a carriage return/line feed sequence after the write operation. The prompt and the response therefore appear on the same line of the terminal.

The WRITEREADX procedure is also useful for issuing control commands to the terminal. For example, you can read the seven-character cursor address from a terminal by issuing an escape sequence as follows:

```
SBUFFER ':=' [%33,"a",%21] -> @S^PTR;
WCOUNT := @S^PTR '-' @SBUFFER;
CALL WRITEREADX(TERMNUM,
                SBUFFER,
                WCOUNT,
                BUFSIZE,
                COUNT^READ);
IF <> THEN ...
```

After the WRITEREADX procedure finishes, SBUFFER contains the seven-character cursor address and 7 is returned in COUNT^READ.

Timing Out Terminal Response

Operations with terminals require human response and therefore can take an indefinite time. You can use the *`timelimit`* parameter of the AWAITIOX procedure to ensure that the operation is completed within a given period of time. To do this, you must have the terminal open to permit nowait I/O.

The following example prompts a user for an account number. If no response is received within five minutes, the user is prompted again:

```
DEFINE FIVE^MINUTES = 30000D#;
LITERAL TIMEOUT = 40;
INT ERROR, .SBUFFER[0:599];
 .
 .
WHILE PROMPT = YES DO
BEGIN
   PROMPT := NO;
   SBUFFER ':=' "Please Enter Account Number" -> @S^PTR;
   CALL WRITEREADX(TERMNUM,
                   SBUFFER,
                   @S^PTR '-' @SBUFFER,
                   BUFSIZE ,
                   COUNT^READ);
   IF <> THEN ...
   CALL AWAITIOX(TERMNUM,
                 !buffer^address!,
                 COUNT^READ,
                 TAG,
                 FIVE^MINUTES);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TERMNUM,
                         ERROR);
      IF ERROR = TIMEOUT THEN PROMPT = YES
      ELSE ....
   END;
END;
```

The above program issues the prompt “Please Enter Account Number” every five minutes until the operator responds.

Echoing Text to the Terminal

When a user types text at a terminal, the text usually appears on the screen as it is typed; that is, the text is echoed. Sometimes it is useful, however, for text to be hidden; for example, when typing in a password. For terminals that are operating in conversational mode and have terminal echo mode configured at system-generation time, you can control whether text is echoed.

Use SETMODE function 20 to programmatically control echo mode. The following call turns off echo mode:

```
LITERAL ECHO^MODE = 20,
        OFF       = 0;
  .
  .
CALL SETMODE(TERMNUM,
             ECHO^MODE,
             OFF);
IF <> THEN ...
```

To turn echo mode back on again:

```
LITERAL ON = 1;
  .
  .
CALL SETMODE(TERMNUM,
             ECHO^MODE,
             ON);
IF <> THEN ...
```

Setting the Transfer Mode

Each terminal has a default transfer mode configured for it at system-generation time. The mode is either conversational or page. Terminals operating in conversational mode transfer each character, as typed, to the system buffers. A file transfer is terminated when a line-termination character (usually a carriage return) is received by the system.

Terminals operating in page mode store each character, as typed, in an internal buffer. The entire block of characters is transferred as one continuous stream. The transfer is usually started when the user presses the ENTER (or SEND or XMIT) key. The file transfer terminates when a page-termination character (usually a carriage return or ETX character) is received by the computer system.

You can override the transfer mode set at system-generation time through a call to the SETMODE procedure with function 8. The following example sets conversational mode:

```
LITERAL MODE           = 8,
        CONVERSATIONAL = 0;
  .
  .
CALL SETMODE(TERMNUM,
             MODE,
             CONVERSATIONAL);
IF <> THEN ...
```

To set page mode:

```
LITERAL PAGE = 1;
 .
 .
CALL SETMODE(TERMNUM,
             MODE,
             PAGE);
IF <> THEN ...
```

Conversational and page modes of operation are described in detail later in this section.

Terminating Terminal Access

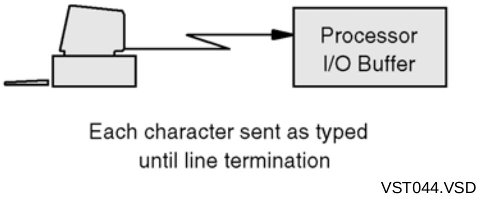
You terminate access to a terminal as you would for any other file, either by stopping the process or by calling the FILE_CLOSE_ procedure:

```
ERROR := FILE_CLOSE_(TERMNUM);
IF ERROR <> 0 THEN ...
```

Communicating in Conversational Mode

When a terminal operates in conversational mode, each character is transferred to the I/O buffer in the computer system as soon as it is typed. The read operation (or file transfer) terminates when a line-termination character is entered at the terminal. Figure 10-1 shows this concept.

Figure 10-1. Conversational Transfer Mode



Once the controller detects the line-termination character and notifies the I/O process, the read operation finishes by transferring the received data into the application buffer through the file-system buffer. The file-system and I/O buffers are released as soon as the read finishes.

Using the Line-Termination Character

A default line-termination character is configured for each terminal at system-generation time. It is usually a carriage return. You can set the line-termination character to any character you like using function 9 of the SETMODE procedure (see Setting the Interrupt Characters for Conversational Mode later in this subsection).

An example explains how the line-termination mechanism works. Suppose a program issues a READX procedure call with a read count of BUFSIZE:

```
CALL READX(TERM^NUM,
           SBUFFER,
           BUFSIZE,
           COUNT^READ);
```

Then the user types the following information:

```
Now is the timeCR
```

```
initial cursor position
```

Following the read operation, the application buffer contains “Now is the time” in its first 17 bytes, and 17 is returned in the COUNT^READ variable.

At the terminal, the carriage return typically triggers a carriage return/line feed sequence. Some terminals provide this feature automatically. For terminals that do not provide this feature, the system sends the terminal a line feed character on receipt of a carriage return. The system is configured to do this. You can use function 7 of the SETMODE procedure to control whether the system sends the line feed character.

The following call causes the file system to send the line feed character automatically on receipt of a carriage return from the terminal:

```
LITERAL LINE^FEED = 7,
        ON = 1;
  .
  .
CALL SETMODE(TERM^NUM,
             LINE^FEED,
             ON);
```

The following call turns off automatic line feed:

```
LITERAL OFF = 0;
  .
  .
CALL SETMODE(TERM^NUM,
             LINE^FEED,
             OFF);
```

If the user responds to the READX call by entering only a carriage return, then the contents of the application buffer remain unchanged, zero is returned in COUNT^READ, and the file system issues a line feed to the terminal:

```
CR
```

`initial cursor position`

Recall that a read operation also terminates when the specified *`read-count`* is satisfied. Suppose your program issues the following procedure call:

```
RCOUNT := 10;
CALL READX(TERMNUM,
           SBUFFER,
           RCOUNT,
           COUNT^READ);
```

Now the user types 10 characters without issuing a carriage return:

```
Now is the
```

`initial cursor position`

"Now is the" is returned in the buffer, and 10 is returned in COUNT^READ. The terminal sends no carriage return, therefore it receives no line feed.

Setting the Interrupt Characters for Conversational Mode

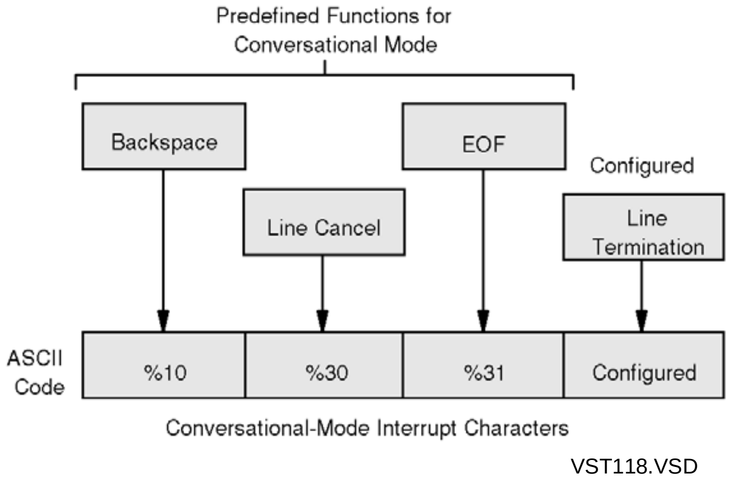
Four programmable interrupt characters are used to cause special actions when encountered. The system-defined default values of these characters are as follows:

Backspace	ASCII code %10
Line cancel	ASCII code %30
End-of-file	ASCII code %31
Line termination	configured

These default values, as summarized in Figure 10-2, apply to a terminal when first opened in conversational mode or when the access mode of the terminal is changed from page mode to conversational mode using SETMODE function 8. The following paragraphs describe the effects of the backspace, line cancel, and end-of-file characters. The line-termination character was described earlier in Using the Line-Termination Character.

Figure 10-2 summarizes the interrupt characters that apply to the conversational mode of operation.

Figure 10-2. Conversational-Mode Interrupt Characters—Default Values


Predefined Functions for Conversational Mode
Backspace
EOF
Configured
Line Cancel
Line Termination
ASCII Code
%10
%30
%31
Configured
Conversational-Mode Interrupt Characters
VST118.VSD

Using the Backspace Character

The backspace character permits the user to back up and then reenter one or more mistyped characters. The specific action involved depends on the type of terminal. Typically, on video terminals the cursor is backspaced one position for each backspace received. On hard-copy devices that can backspace, a line feed and a backspace are issued for the first backspace received, and a single backspace is issued for each subsequent backspace received. On hard-copy devices that do not backspace, a backslash (\) is printed for each backspace entered.

Backspacing is invisible to the application program, because the read operation is not yet complete. In other words, the data eventually returned to the application has already been edited to reflect the changes intended by the backspacing. The terminal I/O process handles the backspacing.

Using the Line-Cancel Character

The line-cancel character permits the user to cancel the current line and reenter it. When the file system receives the line-cancel character, the system writes an “@” character, followed by a carriage return and a line feed (CRLF) to the terminal.

Line cancel is invisible to the application program. When the read from the terminal eventually finishes, everything entered before the line-cancel character has already been discarded and the only data returned to the application are the characters that were typed after the line-cancel character was entered.

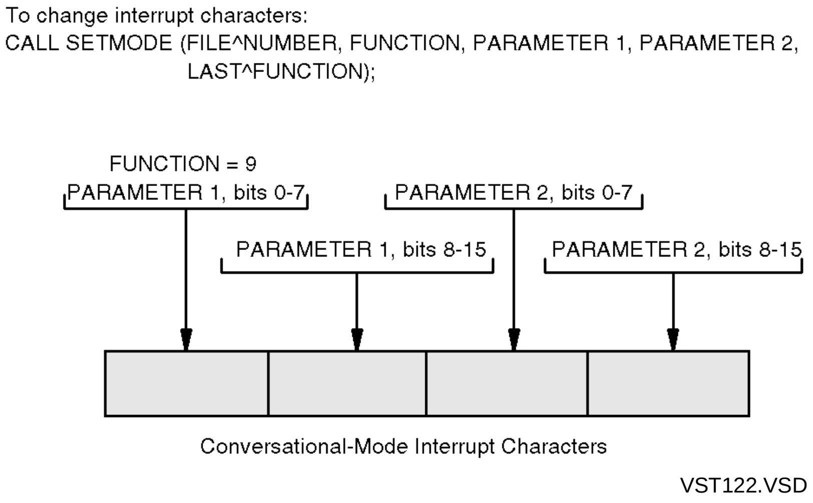
Using the End-of-File Character

The end-of-file character permits a user to signal an application process that no more data will be entered. When the file system receives the end-of-file character, the current file operation is considered to be complete. No data is transferred into the application program's buffer area, the *`count-read`* parameter returns 0, and the condition code indicator is set to CCG. The system writes an "EOF!CRLF" character sequence to the terminal.

Programming the Interrupt Characters

You can change any of the interrupt characters for special applications using SETMODE function 9 as shown in Figure 10-3. The backspace and line-cancel characters are replaced by the upper and lower bytes, respectively, of parameter 1, while the end-of-file and line-termination characters are replaced by the upper and lower bytes, respectively, of parameter 2.

Figure 10-3. Changing Conversational-Mode Interrupt Characters



To change interrupt characters:

```
CALL SETMODE (FILE^NUMBER, FUNCTION, PARAMETER 1, PARAMETER 2,
                    LAST^FUNCTION);
```

Receipt of any interrupt character other than the system-defined interrupt characters always has the same effect, regardless of which interrupt character it replaces:

- The system considers the operation to be complete.
- The application program receives the interrupt character in the buffer along with the line image (if any).
- The *`count-read`* parameter includes the interrupt character.

The following example replaces the configured line-termination character with the line feed character. The other interrupt characters remain unchanged:

```
LITERAL INTERRUPT^CHARACTERS = 9;
 .
 .
PARAM1 ':=' [%10,%30];
PARAM2 ':=' [%31,%12];
CALL SETMODE(TERM^NUM,
             INTERRUPT^CHARACTERS,
             PARAM1,
             PARAM2);
IF <> THEN ...
```

The user now terminates each line with line feed instead of carriage return. The line feed character is always transmitted to the application buffer and is counted in the value returned in the *`count-read`* parameter.

Setting Transparent Mode

You can force the file system to ignore interrupt characters and have them simply passed on to the application as any other character. This is the transparent mode. You select transparent mode by issuing a SETMODE procedure call with function 14. The following procedure call turns on transparent mode:

```
LITERAL TRANSPARENT^MODE = 14,
        ON = 0;
 .
 .
CALL SETMODE(TERMNUM,
             TRANSPARENT^MODE,
             ON);
IF <> THEN ...
```

The following call turns off transparent mode:

```
LITERAL OFF = 1;
 .
 .
CALL SETMODE(TERMNUM,
             TRANSPARENT^MODE,
             OFF);
IF <> THEN ...
```

Once transparent mode is operative, READ and WRITEREAD operations terminate only when the *`read-count`* is satisfied.

Controlling Forms Movement

The SETMODE and CONTROL procedures explicitly control forms movement in conversational mode.

Controlling Spacing

Use SETMODE function 6 to change between single spacing and no spacing when writing data to the terminal. In single spacing the system appends a carriage return/line feed sequence to each write operation. No spacing gives you the option of not sending a carriage return/line feed sequence; if you choose not to send the carriage return/line feed sequence, then successive writes appear on the same line. Single spacing is the default spacing.

The following code turns off single spacing and positions the cursor following the last character written:

```
LITERAL SET^SPACE = 6,
        NO^SPACE  = 0;
  .
  .
CALL SETMODE(TERMNUM,
             SET^SPACE,
             NO^SPACE);
IF <> THEN ...
```

The following code turns single spacing back on again:

```
LITERAL SPACE = 1;
  .
  .
CALL SETMODE(TERMNUM,
             SET^SPACE,
             SPACE);
IF <> THEN ...
```

Another reason for using no spacing would be if you needed to overprint on a hard-copy terminal. By appending a carriage return character to the data to be written, you can cause a carriage return without a line feed:

```
LITERAL SET^SPACE = 6,
        NO^SPACE  = 0;
DEFINE TWO^TENTHS^OF^SECOND = 20D;
STRING .SBUFFER[0:511];
  .
  .
CALL SETMODE(TERMNUM,
             SET^SPACE,
             NO^SPACE);
IF <> THEN ...
```

```
SBUFFER ':=' ["Denote blanks by b.",%015] -> @S^PTR;
CALL WRITEX(TERMNUM,
            SBUFFER,
            @S^PTR ‘-’ @SBUFFER);
IF <> THEN ...

CALL DELAY(TWO^TENTHS^OF^SECOND);

CALL SETMODE(TERMNUM,
             SET^SPACE,
             SPACE);
IF <> THEN ...

SBUFFER ':=' "                 / " -> @S^PTR;
CALL WRITEX(TERMNUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

The example prints the text “Denote blanks by b.” and then overstrikes the “b” with the slash character (/).

Because the application program is supplying the carriage return character, a delay (dependent on the particular terminal involved) might be needed to give the terminal enough time to complete the carriage return operation. You can accomplish this by writing some null characters to the terminal or by calling the DELAY procedure.

Controlling Form Feed

Use CONTROL operation 1 to perform form feed or vertical tabulation operations. The CONTROL parameters for these operations are:

0 for form feed
1 or greater for vertical tabulation

The following example causes an advance to the top of the form for a hard-copy terminal:

```
LITERAL FORMS^CONTROL = 1,
        FORM^FEED     = 0;
 .
 .
CALL CONTROL(TERMNUM,
             FORMS^CONTROL,
             FORM^FEED);
```

The system automatically delays subsequent access to the same terminal for a configured period of time after performing forms control through the CONTROL procedure.

If the configured delay is not suitable, the application program can issue a form feed (%14) or vertical tabulation (%13) character through a WRITE procedure call.

However, in this case you must delay the application program to permit the forms movement to finish:

```
DEFINE TWO^SECONDS = 200D;
 .
 .
CALL WRITEX(TERMNUM,%014,1);
IF <> THEN...

CALL DELAY(TWO^SECONDS);
```

The application suspends itself for two seconds after sending the form-feed character to the terminal.

Communicating in Page Mode

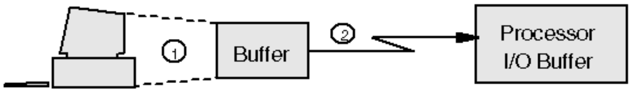
Normally, terminals operating in page mode store each character in display memory in the terminal as it is typed. Display memory is divided into logical pages consisting of 1920 bytes. An entire page of display memory is sent to the computer system at once as a series of write operations of 256 bytes each. The transfer begins when the user presses the ENTER (or SEND or XMIT) key. A file transfer terminates when the computer system receives a page-termination character (typically a carriage return or ETX character). See Figure 10-4.

For terminals that operate in pseudopolled page mode, the transfer mechanism is different. Here, the sequence of events is as follows:

1. The user types a block of characters.
2. The user presses the ENTER key, informing the computer system that the terminal is ready to send a block of information. (The block of information is not sent yet.)
3. The system responds by sending a “trigger” character back to the terminal.
4. The terminal responds to the trigger by sending the complete block of information to the I/O buffer in the computer system.

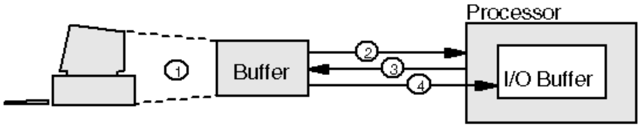
Figure 10-4. Page Transfer Mode

Page Mode



① Buffer fills with characters as typed.
② Block sent when XMIT typed.
Read until line termination.

Pseudopolled Page Mode



① Buffer fills with characters as typed.
② Control characters sent when XMIT typed.
③ Trigger
④ All characters sent up to
page-termination character.

VST045.VSD

Using the Page-Termination Character

A page-termination character is configured at system-generation time for each page-mode terminal. You can set the page-termination character to any character you like using function 9 of the SETMODE procedure (see Setting the Interrupt Characters for Page Mode below).

The page-termination character, when received from a terminal, signals the computer system that the current page transfer is complete. When the read operation is complete, the page of data occupies the buffer specified by the application. This buffer does not contain the page-termination character unless the buffer would otherwise contain an odd number of bytes. The page-termination character is not counted in the *`count-read`* parameter returned by the read operation.

The system does **not** issue a carriage return/line feed sequence to the terminal on receipt of the page-termination character.

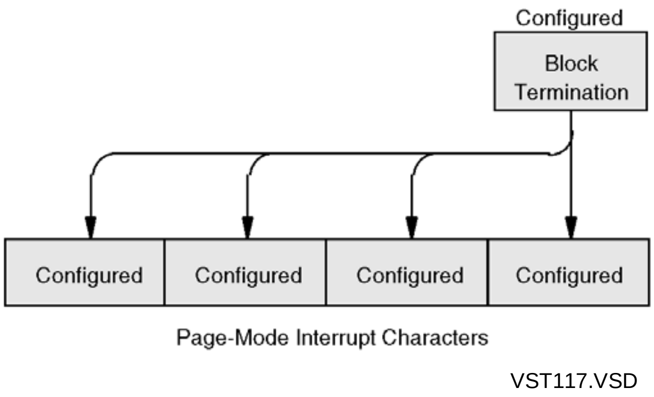
As with conversational mode, the read operation automatically finishes if the *`read-count`* specified in the read operation is satisfied.

Setting the Interrupt Characters for Page Mode

Initially, the only valid interrupt character is the page-termination character. A shown in Figure 10-5, all four interrupt characters that apply to page mode are set to the configured page-termination character.

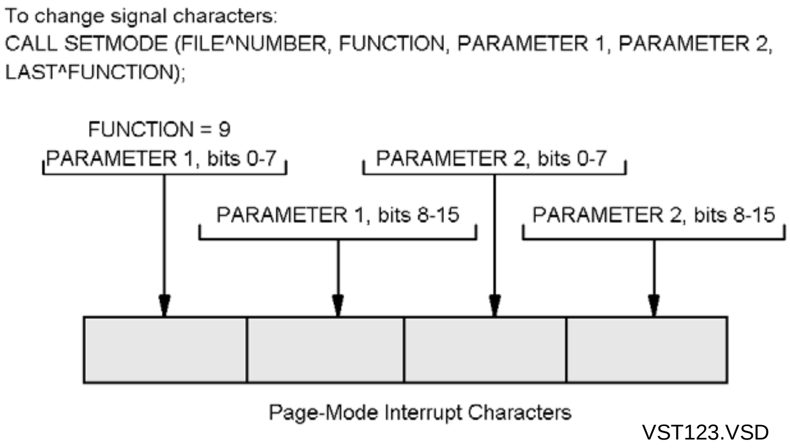
These interrupt characters apply to each page-mode terminal when the terminal is first opened. The same default values are restored when you dynamically change from conversational mode to page mode.

Figure 10-5. Page-Mode Interrupt Characters—Default Values


Configured
Block Termination
Configured
Configured
Configured
Configured
Page-Mode Interrupt Characters
VST117.VSD

You can change the page-mode interrupt characters to other values by using SETMODE function 9 with the terminal in page mode, as shown in Figure 10-6. You must provide all four interrupt characters in parameters 1 and 2 of the SETMODE call.

Figure 10-6. Changing Page-Mode Interrupt Characters


To change signal characters:
CALL SETMODE (FILE^NUMBER, FUNCTION, PARAMETER 1, PARAMETER 2, LAST^FUNCTION);
FUNCTION = 9
PARAMETER 1, bits 0-7
PARAMETER 2, bits 0-7
PARAMETER 1, bits 8-15
PARAMETER 2, bits 8-15
Page-Mode Interrupt Characters
VST123.VSD

Receipt of any interrupt character other than the configured page-termination character has the following effect:

- The system considers the operation to be complete.
- The application program receives the page-termination character in the application buffer along with the page image (if any).
- The *`count-read`* parameter returned by the read operation includes the interrupt character.

The following example shows the action of the interrupt characters when you dynamically change from conversational mode to page mode and then back to conversational mode. The configured line-termination character is a carriage return; the configured page-termination character is also a carriage return. The terminal is configured as a conversational-mode terminal.

```
LITERAL CHANGE^MODE = 8,   !function for SETMODE; change
                           !transfer mode
        CONV^MODE = 0,     !parameter for SETMODE;
                           !conversational mode
        PAGE^MODE = 1,     !parameter for SETMODE; page mode
        SET^INTCHARS = 9,  !function for SETMODE; set
                           !interrupt characters
        BS^CAN = %004030,  !backspace, line cancel
        HT^CR = %004415,   !horizontal tab, carriage return
        ETX^EOT = %001404; !end-of-text, end-of-transmission
  .
  .
```

First, open the terminal.

```
ERROR := FILE_OPEN_(TERM^NAME,
                    TERM^NUM);
IF ERROR <> 0 THEN ...
  .
  .
```

The terminal opens in conversational mode because it is configured that way. The default interrupt characters are now in force: backspace, line cancel, end of file, carriage return.

Now call the SETMODE procedure with function 9 to change the interrupt characters for conversational mode:

```
CALL SETMODE(TERM^NUM,
             SET^INTCHARS,
             BS^CAN,
             HT^CR);
IF <> THEN ...
  .
  .
```

The conversational-mode interrupt characters are now set to backspace, line cancel, horizontal tab, and carriage return.

Now call SETMODE again, but to change the mode to page mode:

```
CALL SETMODE(TERM^NUM,CHANGE^MODE,PAGE^MODE);
IF <> THEN ...
 .
 .
```

The terminal is now operating in page mode with all four interrupt characters set to carriage return (the configured default).

Now call SETMODE again, this time to change the interrupt characters for page mode:

```
CALL SETMODE(TERM^NUM,
             SET^INTCHARS,
             ETX^EOT,ETX^EOT);
IF <> THEN ...
 .
 .
```

The interrupt characters for page mode are now end-of-text, end-of-transmission, end-of-text, and end-of-transmission.

Now call SETMODE again to change the transfer mode back to conversational:

```
CALL SETMODE(TERM^NUM,
             CHANGE^MODE,
             CONV^MODE);
IF <> THEN ...
 .
 .
```

The interrupt characters are restored to their initial values: backspace, line cancel, end of file, and carriage return.

Note. When setting interrupt characters with SETMODE, you must specify all four characters. If you do not need four interrupt characters, some must be duplicated.

As with conversational-mode terminals, you can force the file system to ignore interrupt characters by turning on transparent mode using SETMODE function 14. Refer to Setting Transparent Mode under Setting the Interrupt Characters for Page Mode earlier in this section.

Communicating With Pseudopolled Terminals

Recall that pseudopolled terminals receive a trigger from the computer system after the terminal is ready to send the page of data. This trigger may be automatically supplied by the file system, or it may be done by the application program. Pseudopolled terminals are always configured during system generation either to receive an automatic trigger or to have the trigger sent by the application.

The advantage of having the system handle triggering is that the operation is invisible to the application. The automatic triggering applies only when a READ[X] procedure is issued to the terminal. It does not apply to WRITEREAD[X], so that the WRITEREAD[X] procedure can be used for operations such as cursor sensing.

The advantage of having the application program handle triggering is that only one word of buffer space is used while the user enters information. The buffer space is allocated after the user presses the ENTER key. (Terminals operating in normal page mode require that the entire system buffer space be allocated during the wait for a transfer to take place.)

Here's how application triggering works. The application program initiates a read operation of one character to the pseudopolled terminal. This read waits for the ready character.

```
RCOUNT := 1;
CALL READX(TERM^NUM,
           SBUFFER,
           RCOUNT,
           COUNT^READ);
```

Reading one byte causes one byte of system buffer space to be allocated. The user types in the page of text, then presses the ENTER key. Pressing ENTER causes a ready character (for example, device-control-2) to be sent to the computer system, causing the read to finish.

The application then issues the trigger character (for example, device-control-1) to the terminal and issues a read of 600 characters. To ensure that the system is ready to start reading when the terminal starts transmitting, you must combine both operations into one using WRITEREADX:

```
SBUFFER := %21;                  !device-control-1
WCOUNT := 1;
RCOUNT := 600;
CALL WRITEREADX(TERM^NUM,
                SBUFFER,
                WCOUNT,
                RCOUNT,
                COUNT^READ);
```

This call to WRITEREADX causes 300 words (600 bytes) of system buffer space to be allocated. Sending the device-control-1 character to the terminal causes the terminal to send a page of information to the computer system. The page is returned to the application process in SBUFFER, and the actual number of bytes read is returned in COUNT^READ. (As with any file-system operation, the system buffer space is deallocated after the read finishes.)

Managing the BREAK Key

The file system enables a user to signal a process by pressing the BREAK key. A common example of the use of the BREAK key to signal a process is in the TACL process: if an application started from a TACL prompt does not perform its own BREAK handling, pressing the BREAK key while the application process is running returns control of the terminal to the TACL process.

An application that performs its own BREAK processing can be interrupted from the terminal without periodically checking the terminal for input; instead, the application simply checks the $RECEIVE file for system message -105 (Break-on-Device message).

You can use BREAK in conversational or page mode.

To write programs that manage the BREAK key, you need to perform some of the following tasks:

- Enable BREAK by taking ownership of the BREAK key
- Receive and process message -105 (the Break-on-Device message)
- Reestablish BREAK ownership after receiving the Break-on-Device message
- Establish BREAK mode
- Interpret BREAK-related errors

The above tasks are outlined below. The remainder of this subsection describes in detail how to perform these tasks.

To enable BREAK and perform its own BREAK handling, your program must take ownership of the BREAK key. You should be aware of the following:

- You establish BREAK ownership using either SETPARAM function 3 or SETMODE function 11. We recommend using the SETPARAM procedure because only that procedure allows you specify a break tag value so that you can distinguish between subdevices when a BREAK occurs.
- Only one process can own the BREAK key at a time.
- If the terminal was opened by the backup process of a process pair, the backup process automatically becomes the BREAK owner if its primary process fails while owning BREAK.
- If BREAK is not enabled, then the BREAK key is ignored.
- If the process that owns BREAK is deleted or fails, BREAK ownership is lost. That is, no process is informed if the BREAK key is pressed.

To receive system message -105 (the Break-on-Device message), your process must have BREAK enabled. Receiving the Break-on-Device message indicates that the BREAK key has been pressed.

See the *Guardian Procedure Errors and Messages Manual* for details on the Break-on-Device message structure.

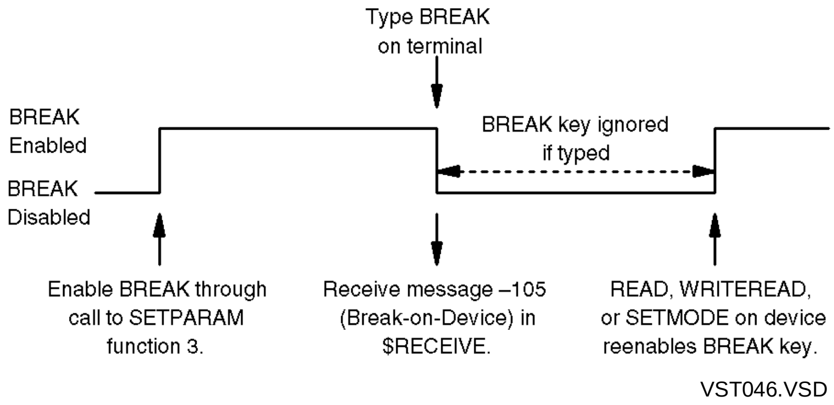
To reestablish BREAK ownership after receiving the Break-on-Device message, your process must either issue a READ or WRITEREAD procedure call to the terminal or reissue the SETPARAM or SETMODE request that establishes BREAK ownership. When the BREAK key is pressed, the BREAK feature is no longer enabled; if the BREAK key is pressed again, it is ignored. You should therefore reestablish BREAK ownership.

To establish BREAK mode for the terminal, you need to use SETMODE function 12. Once in BREAK mode, only operations that are associated with BREAK are allowed to access the terminal.

To interpret BREAK-related errors, check for error 110 (only BREAK access permitted) and error 111 (operation aborted because of BREAK). Any process using the same terminal as the TACL process or other process that handles the BREAK key must check for these errors. See Recovering From Errors later in this section.

Figure 10-7 shows the sequence of events when establishing and reestablishing BREAK ownership.

Figure 10-7. Enabling BREAK


Type BREAK
on terminal
BREAK
Enabled
BREAK
Disabled
BREAK key ignored
if typed
Enable BREAK through
call to SETPARAM
function 3.
Receive message –105
(Break-on-Device) in
$RECEIVE.
READ, WRITEREAD,
or SETMODE on device
reenables BREAK key.
VST046.VSD

Taking BREAK Ownership

When a process takes BREAK ownership for a terminal from another process (for example, a TACL process), the application process should identify the process that currently owns BREAK and get the BREAK mode of the current owner. You can do this by specifying the *last-param-array* to the SETPARAM procedure call:

```
LITERAL SET^BREAK^FUNCTION = 3,
        NORMAL^MODE        = 0,
        TAKE^BREAK         = 1;

INT LAST^PARAM^ARRAY[0:3];
INT PARAM^ARRAY[0:3];
  .
  .
PARAM^ARRAY[0] := TAKE^BREAK;
PARAM^ARRAY[1] := NORMAL^MODE;
PARAM^ARRAY[2] := 0;
PARAM^ARRAY[3] := 0;
PARAM^COUNT := 8;
CALL SETPARAM(TERM^NUM,
              SET^BREAK^FUNCTION,
                  PARAM^ARRAY,
                  PARAM^COUNT,
                  LAST^PARAM^ARRAY,
                  LAST^PARAM^COUNT);
```

The first word of the *`last-param-array`* contains an internally defined integer that identifies the current owner of BREAK. The second word contains an indication of the BREAK mode.

The *`last-param-count`* returns the length of the *`last-param-array`* value in bytes, and is always 8 for function 3.

Releasing BREAK Ownership

When your application no longer wants to receive Break-on-Device messages, it should reenable BREAK for the last owner.

To return BREAK ownership to the previous BREAK owner, you simply supply the SETPARAM (or SETMODE) procedure with the internal process identifier and BREAK mode that you acquired in the call that took BREAK ownership:

```
CALL SETPARAM(TERM^NUM,
              SET^BREAK^FUNCTION,
              LAST^PARAM^ARRAY,
              LAST^PARAM^COUNT);
```

In other words, you use the *`last-param-array`* and *`last-param-count`* values returned by the previous SETPARAM call as the *`param-array`* and *`param-count`* parameters to this call.

Selecting BREAK Mode

Although several processes may have access to a terminal, a process can gain exclusive access to that terminal when BREAK is pressed. Such a process is executing in BREAK mode. You establish BREAK mode at the same time you take BREAK ownership.

When a process executes in BREAK mode, it can communicate with the terminal using only operations that have BREAK access. Once BREAK access is established, the process has exclusive access to the terminal.

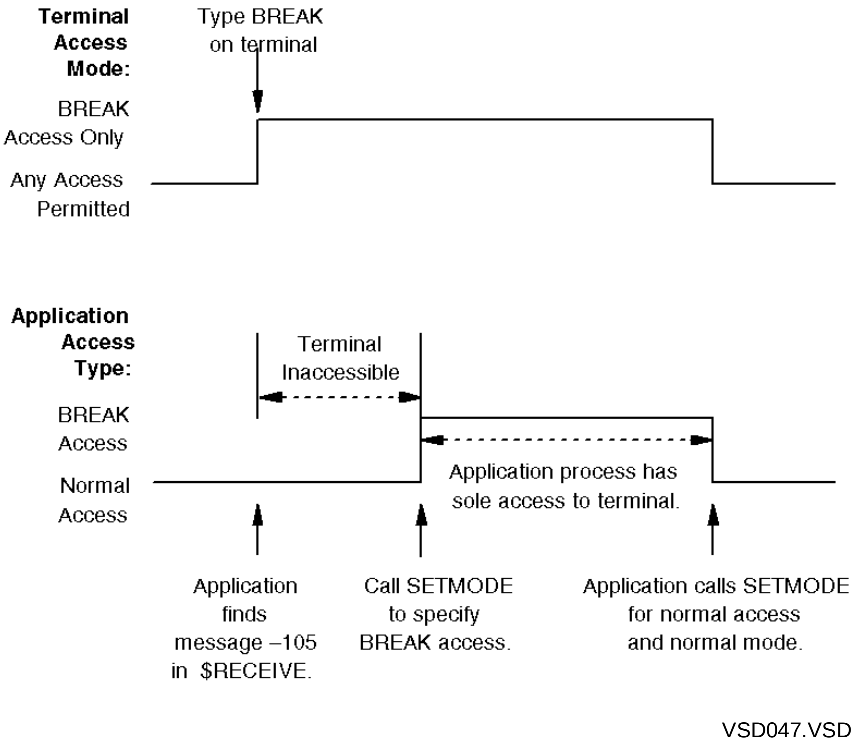
The following steps are involved in using BREAK mode:

1. Enable BREAK and establish BREAK mode using either SETPARAM function 3 or SETMODE function 11.
2. Set BREAK access using SETMODE function 12.
3. Relinquish BREAK mode and BREAK access using SETMODE function 12.
4. Return ownership to the previous BREAK owner using SETPARAM function 3 or SETMODE function 11.

These steps are described in detail in the following paragraphs.

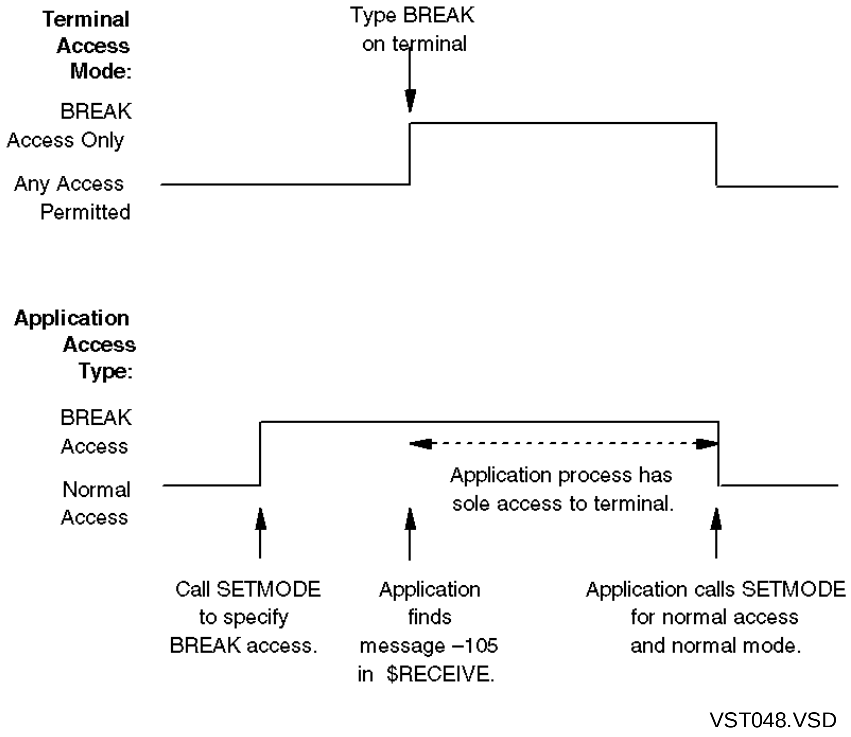
Your program can establish BREAK access before or after the user presses the BREAK key. If you establish BREAK access after pressing the BREAK key, then the terminal is inaccessible between pressing the BREAK key and establishing BREAK access, as shown in Figure 10-8.

Figure 10-8. BREAK Access Established After Pressing the BREAK Key


Terminal Access Mode:
Type BREAK on terminal
BREAK Access Only
Any Access Permitted
Application Access Type:
Terminal Inaccessible
BREAK Access
Normal Access
Application process has sole access to terminal.
Application finds message –105 in $RECEIVE.
Call SETMODE to specify BREAK access.
Application calls SETMODE for normal access and normal mode.
VSD047.VSD

If you establish BREAK access before pressing the BREAK key, then the terminal becomes accessible immediately after the BREAK key is pressed, as shown in Figure 10-9.

Figure 10-9. BREAK Access Established Before Pressing the BREAK Key


Terminal Access Mode:
BREAK Access Only
Any Access Permitted
Type BREAK on terminal
Application Access Type:
BREAK Access
Normal Access
Application process has sole access to terminal.
Call SETMODE to specify BREAK access.
Application finds message –105 in $RECEIVE.
Application calls SETMODE for normal access and normal mode.
VST048.VSD

Establishing BREAK Mode

To establish BREAK mode, you must specify BREAK mode when enabling BREAK. Doing this tells the system to put the terminal into BREAK mode when the BREAK key is pressed. Once BREAK mode is established, only file operations having BREAK access are allowed access to the terminal.

The following example enables BREAK. The second word of the *`param-array`* specifies BREAK mode:

```
LITERAL SET^BREAK^FUNCTION = 3,
        BREAK^MODE         = 1,
        TAKE^BREAK         = 1;
INT    .PARAM^ARRAY[0:3];
INT     PARAM^COUNT;
INT    .LAST^PARAM^ARRAY[0:3];
INT     LAST^PARAM^COUNT;
  .
  .
PARAM^ARRAY[0] := TAKE^BREAK;
PARAM^ARRAY[1] := BREAK^MODE;
PARAM^ARRAY[2] := 0;
PARAM^ARRAY[3] := 0;
PARAM^COUNT := 8;
CALL SETPARAM(TERM^NUM,
              SET^BREAK^FUNCTION,
              PARAM^ARRAY,       !word 1 specifies BREAK mode
              PARAM^COUNT
              LAST^PARAM^ARRAY,
              LAST^PARAM^COUNT);
```

Establishing BREAK Access

When the system puts the terminal into BREAK mode, any operations on the terminal, even those from the owner of BREAK, are rejected unless the program making the access has put itself into BREAK access mode. By convention, only the owner of BREAK is supposed to do that, hence this mechanism ensures that when the terminal user presses BREAK, the BREAK owner can respond.

Use SETMODE function 12 to establish BREAK access to the terminal. Once BREAK access is established, the application process can communicate with the terminal in the usual way.

The following statement establishes BREAK access:

```
LITERAL BREAK^ACCESS =  1;
LITERAL SET^ACCESS   = 12;
  .
  .
CALL SETMODE(HOME^TERM^NUM,
             SET^ACCESS,
             !param1!,
             BREAK^ACCESS);
```

Reestablishing Normal Access and Normal Mode

Another call to SETMODE function 12 relinquishes BREAK access and BREAK mode, returning the terminal status to normal access and normal mode. You achieve this by setting *`parameter-1`* and *`parameter-2`* to zero:

```
LITERAL NORMAL^ACCESS = 0,
        NORMAL^MODE   = 0;
  .
  .
CALL SETMODE(HOME^TERM^NUM,
             SET^ACCESS,
             NORMAL^MODE,
             NORMAL^ACCESS);
```

All types of access are now permitted to the terminal.

Returning BREAK to the Previous Owner

Finally, you should return BREAK to the previous owner using either SETPARAM function 3 or SETMODE function 11:

```
CALL SETPARAM(TERM^NUM,
              SET^BREAK^FUNCTION,
              LAST^PARAM^ARRAY,
              LAST^PARAM^COUNT);
```

Using BREAK Mode: An Example

In the following example, the SETPARAM call that establishes BREAK ownership also sets BREAK mode and saves the identification and BREAK mode of the previous owner. After enabling BREAK, this example checks $RECEIVE for Break-on-Device messages. Normally, the process loops doing some computation, without any interaction with the terminal user. Part of that loop checks $RECEIVE for the Break-on-Device message. On receipt of a Break-on-Device message, the main procedure calls the BREAK^IT procedure to process the Break-on-Device message.

```
?INSPECT, SYMBOLS

!Literals:
LITERAL SET^BREAK^FUNCTION =   3,
        NORMAL^MODE        =   0,
        NEW^OWNER          =   1,
        BREAK^ACCESS       =  12,
        BREAK^ACCESS^ON    =   1,
        BREAK^ACCESS^OFF   =   0,
        BREAK^MODE         =   1,
        NOWAIT             =   1,
        MAXLEN             = 256;
```

```
!Global variables:

STRING .HOME^TERM[0:MAXLEN -1],     !terminal file name
       .RECV^FILE[0:7]
                    := "$RECEIVE";  !$RECEIVE file name

INT  HOME^TERM^NUM,                 !terminal number
    .BUFFER[0:127],                 !buffer for terminal I/O
     RCOUNT,                        !max bytes read
     WCOUNT,                        !count of bytes to write
     ERROR,                         !file-system error number
     BYTES^READ,                    !number of bytes read
    .PARAM^ARRAY[0:3],              !input to SETPARAM
     PARAM^COUNT,                   !length of PARAM^ARRAY
    .LAST^PARAM^ARRAY[0:3],         !values for last owner
     LAST^PARAM^COUNT,              !size of last-param-array
    .RECV^BUF[0:66],                !buffer for $RECEIVE
                                    ! messages
     RECV^NUM,                      !file number for $RECEIVE
     LOOP, I,J;                     !computation variables

STRING .FIRST^BYTE := @BUFFER '<<' 1; !string pointer to
                                      ! BUFFER

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS(PROCESS_GETINFO,FILE_OPEN_,
?               WRITE,READ,WRITEREAD,FILE_GETINFO_,
?               PROCESS_STOP_,DEBUG,SETPARAM,
?               AWAITIO,INITIALIZER,SETMODE)
?LIST
?NOMAP,NOCODE

!-------------------------------------------------------------!
Procedure to process Break-on-Device message.  This
! procedure prompts the user for input and then echoes the
! input back to the terminal.  If the user types "exit," then
! the process terminates.  If the user types "resume," then
! the process returns to computational mode until the BREAK
! key is pressed again.  For any other user response, this
! procedure displays the prompt.
!-------------------------------------------------------------

PROC BREAK^IT;
BEGIN
   WHILE 1 DO
   BEGIN

   !  Establish BREAK access:

      CALL SETMODE(HOME^TERM^NUM,BREAK^ACCESS,
                   !param1,
                   BREAK^ACCESS^ON);
      !  Prompt the user to enter a string of characters:

         WCOUNT := 2;
         BUFFER ':=' "? ";
         RCOUNT := 128;
         CALL WRITEREAD(HOME^TERM^NUM,BUFFER,WCOUNT,
                        RCOUNT,BYTES^READ);
         IF <> THEN
         BEGIN
            CALL FILE_GETINFO_(HOME^TERM^NUM,ERROR);
            CALL DEBUG;
         END;

      !  If the user enters "exit" then terminate:

         IF FIRST^BYTE = "exit" THEN
         BEGIN
            CALL SETMODE(HOME^TERM^NUM,BREAK^ACCESS,NORMAL^MODE,
                         BREAK^ACCESS^OFF);
            LAST^PARAM^COUNT := 8;
            CALL SETPARAM(HOME^TERM^NUM,SET^BREAK^FUNCTION,
            LAST^PARAM^ARRAY,LAST^PARAM^COUNT);
            CALL PROCESS_STOP_;
         END;

      !  If the user types "give break", return BREAK to
      !  previous owner and resume:

         IF FIRST^BYTE = "give break" THEN
         BEGIN
            CALL SETMODE(HOME^TERM^NUM,BREAK^ACCESS,NORMAL^MODE,
                         BREAK^ACCESS^OFF);
            LAST^PARAM^COUNT := 8;
            CALL SETPARAM(HOME^TERM^NUM,SET^BREAK^FUNCTION,
            LAST^PARAM^ARRAY,LAST^PARAM^COUNT);
            RETURN;
         END;

      !  If the user enters "resume" then give up BREAK access
      !  and return to computational mode:

         IF FIRST^BYTE = "resume" THEN
         BEGIN
            CALL SETMODE(HOME^TERM^NUM,BREAK^ACCESS,
                         !param1!,
                         BREAK^ACCESS^OFF);
            RETURN;
         END;

         !Otherwise echo the typed string to the terminal then
         !loop to prompt for more input:
         CALL WRITE(HOME^TERM^NUM,BUFFER,BYTES^READ);
      END;
   END;
```

```
!------------------------------------------------------------
! Main procedure does computation without terminal
! interaction.  The procedure checks $RECEIVE periodically
! for a Break-on-Device message and then calls BREAK^IT to
! process the message.
!------------------------------------------------------------

PROC TERMS MAIN;
BEGIN

!  Process the Startup message:

   CALL INITIALIZER;

!  Open the terminal file:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name:maxlen!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         HOME^TERM:MAXLEN,
                         LENGTH);
   ERROR :=  FILE_OPEN_(HOME^TERM:LENGTH,HOME^TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the $RECEIVE files for nowait I/O

   LENGTH := 8;
   ERROR := FILE_OPEN_(RECV^FILE:LENGTH,RECV^NUM,
                       !access!,
                       !exclusion!,
                       NOWAIT);
   IF ERROR <>0 THEN CALL PROCESS_STOP_;

!  Enable BREAK:

   PARAM^ARRAY[0] := NEW^OWNER;
   PARAM^ARRAY[1] := BREAK^MODE;
   PARAM^ARRAY[2] := 0;
   PARAM^ARRAY[3] := 0;
   PARAM^COUNT := 8;
   LAST^PARAM^COUNT := 8;
   CALL SETPARAM(HOME^TERM^NUM,SET^BREAK^FUNCTION,
                 PARAM^ARRAY,PARAM^COUNT,
                 LAST^PARAM^ARRAY,LAST^PARAM^COUNT);
```

```
!  Loop indefinitely, checking for Break-on-Device message:

   WHILE 1 = 1 DO
   BEGIN

   !  Issue a nowait read on $RECEIVE:

      CALL READ(RECV^NUM,RECV^BUF,132);
      ERROR := 0;
      LOOP := 40;

   !  Loop until nowait read finishes:

      WHILE LOOP = 40 DO
      BEGIN

      !Check for completion of read operation.  Return
      !immediately if incomplete:

         CALL AWAITIO(RECV^NUM,
                      !buffer^address!,
                      BYTES^READ,
                      !tag!,
                      0D);
         IF = THEN
         BEGIN

         !  Process user message
                .
                .
         END;

      !  Check if system message:

         IF > THEN
         BEGIN

         !  Check if Break-on-Device message:

            IF RECV^BUF = -105 THEN
            CALL BREAK^IT
            ELSE
            BEGIN

            !  Process other system message
                   .
                   .
            END;
         END

      !  Else AWAITIO returned with an error:

         ELSE CALL FILE_GETINFO_(RECV^NUM,ERROR);
         LOOP := ERROR;
      !  Do some computation -- this code could be any non-
      !  interactive task:

         J := 0;
         WHILE J < 2000 DO
         BEGIN
            I := 0;
            WHILE I < 2000 DO
            I := I + 1;
            J := J + 1;
         END;
      END;
   END;
END;
```

Recovering From Errors

For terminals, error recovery depends on the specific error. Possible errors can be categorized as follows

- Errors that can be retried indefinitely
- Errors that should be retried but only a limited number of times
- Errors that need special attention
- Errors for which retrying the operation makes no sense

The following errors can be retried indefinitely. This can be important because in some situations a read operation, for example, might not complete for several days:

112	Operation preempted by operator message
230	CPU power failed, then restored
231	Controller power failed then restored
246-249	Expand errors

Errors 201 through 229 should be retried a limited number of times. These errors indicate an error in the path to the terminal. Typically, you should retry these errors between 3 and 10 times.

The following errors often need special attention:

30-39	Temporary lack of resources
40	Operation timed out
110 and 111	BREAK errors
112	Preempted by operator message
140	Modem error

The following paragraphs describe the effects of these errors. At the end of this subsection is a sample program for dealing with terminal errors.

For all errors, you can get a short description of the error using the TACL ERROR command with the error number as parameter. For more detailed information on the error, refer to the *Guardian Procedure Errors and Messages Manual*.

Recovering From Errors That Indicate a Temporary Lack of Resources

Errors in the range 30 through 39 indicate that some resource is lacking, such as file system or I/O process buffer space, file-system control blocks, or process control blocks. These errors can be retried a limited number of times after a short delay.

Recovering From an “Operation Timed Out” Error

Error 40 indicates that the user did not respond to the application within the period specified in the AWAITIO procedure call. Any data entered before the timeout occurred is lost. You should therefore send a message to the user to reenter the data.

Recovering From a BREAK Error

Pressing BREAK on a terminal where BREAK is enabled can cause an application process to receive either of two errors:

Error 110	only BREAK access permitted
Error 111	operation aborted because of BREAK

The action taken for these errors depends on whether the process receiving the error is the one with BREAK enabled (the process that receives the Break-on-Device message).

Error 110 indicates that the BREAK key was pressed and that BREAK mode was specified when BREAK was enabled (by SETPARAM function 3 or SETMODE function 11). The terminal is inaccessible until the process calls SETMODE function 12 to allow normal access to the terminal.

If the process receiving error 110 is not the one that enabled BREAK, then the operation should be retried periodically. If the process has BREAK enabled, it should check $RECEIVE for the system Break-on-Device message and take appropriate action.

Error 110 implies that no data was transferred.

Error 111 indicates that BREAK was pressed while the current file operation was taking place. This error indicates that data may have been lost.

If the process receiving error 111 is not the one that enabled BREAK, then you should retry the operation. If a write operation was being performed, then the write can simply be retried. If a read operation was being performed, then a message should be sent advising the user to retype the last entry before retrying the read.

Keep in mind, however, that if more than one process is accessing a terminal and the BREAK feature is used, only BREAK access should be allowed after BREAK is pressed. Therefore, subsequent retries are rejected with error 110 until normal access is permitted.

If either of these errors is received by a process not having BREAK enabled, the process should suspend itself for some short period (such as 1 second) before retrying the operation. You can do this by calling the process-control procedure DELAY. If you use the FILEERROR procedure to retry the failed operation, the delay is applied automatically.

If the process has BREAK enabled, then you should check $RECEIVE for the system Break-on-Device message and then take appropriate action.

Responding to Operator Preemption

Error 112 can occur only if the application process is using the same terminal as the active operator console device. If the application process is reading from the terminal (using either READ[X] or WRITEREAD[X]) and a message is sent to the operator, the read operation is aborted and the operator message is written (that is, operator messages have a higher priority). Any data entered when the preemption takes place is lost. The application process should therefore send a message to the user to reenter the data.

Recovering From a Modem Error

Error 140 occurs if the carrier signal to the modem was lost. The carrier loss may be a permanent or momentary loss. In either case, it must be assumed that the data was lost.

The first time error 140 occurs, you should send a message to the user to try entering the data again. If error 140 recurs after you send this message, then the connection with the remote terminal is lost. You should then call the CONTROL procedure once to disconnect the modem (operation 12) and then again to wait for modem reconnection (operation 11).

Recovering From a Path Error

The application should count how many times path errors 201 through 229 occur on a particular file. Such an error indicates that one path to the associated device has failed. If the error recurs when you try the operation again, then both paths have failed and the device is no longer accessible. If the retry succeeds, then either the alternate path was successful or the process may have created another backup (because of a IPU reload or an action by the application program).

If an error 210 through 231 occurs, then the operation failed at some indeterminate point. If reading, you should send a message to the user to reenter the data. Your application should then try the read operation again.

Recovering From Errors: A Sample Program

The TERM^IO procedure shown in the following example provides a simple way of handling terminal I/O errors. It divides all errors into those that can be indefinitely retried and those that should not be indefinitely retried. For indefinitely retryable errors, the procedure keeps repeating the operation as many times as necessary until the operation is successful.

For all other errors, the procedure attempts the operation up to five times before giving up. For simplicity, the procedure retries the operation for errors that never go away on a retry; it does no harm.

The procedure assumes the following about the process:

- The process does not set a timeout and therefore will never receive a timeout error (error 40).
- The process does not enable BREAK mode
- If there are any modem connections, no attempt is made to wait for reconnection of the modem following a permanent modem disconnection

```
?INSPECT, SYMBOLS, NOCODE
?NOLIST
?SOURCE $TOOLS.ZTOOLD04.ZSYSTAL;
?LIST

!  Literals:

LITERAL RETRY^LIMIT = 5;
LITERAL YES         = 1;
LITERAL NO          = 0;

!  Global variables:

INT     TERM^NUM;
STRING .SBUFFER;
INT     RCOUNT;
STRING .S^PTR;


?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,WRITEREADX,
?                          FILE_GETINFO_,WRITEX,DELAY,
?                          PROCESS_STOP_);
?LIST

.
.
.
```

```
!------------------------------------------------------------------
! Procedure to perform terminal I/O
!------------------------------------------------------------------

PROC TERM^IO;
BEGIN
   INT  WCOUNT;
   INT  COUNT^READ;
   INT  DONE := NO;
   INT  RETRY^COUNT;
   INT  ERROR;

!  Loop until file operation successful:

   WHILE DONE = NO DO
   BEGIN

   !  Set flag for success, initialize retry count, and prompt
   !  for input:

      DONE := YES;
      RETRY^COUNT := 0;
      SBUFFER ':=' "APPL1>" -> @S^PTR;
      WCOUNT := @S^PTR '-' @SBUFFER;
      CALL WRITEREADX(TERM^NUM,SBUFFER,WCOUNT,RCOUNT,
                      COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TERM^NUM,ERROR);
         IF ERROR = 112
         OR ERROR = 200
         OR ERROR = 230
         OR ERROR = 231
         OR ERROR = 240
         OR ERROR = 241
         OR ERROR = 246
         OR ERROR = 248
         OR ERROR = 249
         THEN

      !  Retry until successful:

         BEGIN
            SBUFFER ':=' "Terminal I/O error: trying again"
                         -> @S^PTR;
            WCOUNT := @S^PTR '-' @SBUFFER;
            CALL WRITEX(TERM^NUM,SBUFFER,WCOUNT);
            CALL DELAY(100D);
            DONE := NO;
         END
         ELSE
      !  Retry up to RETRY^LIMIT:

         BEGIN
            RETRY^COUNT := RETRY^COUNT + 1;
            IF RETRY^COUNT < RETRY^LIMIT THEN

         !  Retry limit not yet reached, so try again after one
         !  second delay:

            BEGIN
               SBUFFER ':=' "Terminal I/O error: trying again"
                            -> @S^PTR;
               WCOUNT := @S^PTR '-' @SBUFFER;
               CALL WRITEX(TERM^NUM,SBUFFER,WCOUNT);
               CALL DELAY(100D);
               DONE := NO;
            END
            ELSE

         !  Retry limit reached.  Stop the process:

            BEGIN
               SBUFFER ':='
                       "Terminal I/O error: operation failed"
                       -> @S^PTR;
               WCOUNT := @S^PTR '-' @SBUFFER;
               CALL WRITEX(TERM^NUM,SBUFFER,WCOUNT);
               CALL PROCESS_STOP_;
            END;
         END;
      END

      ELSE
      CALL PROCESS^INPUT(COUNT^READ);
   END;
END;
```

11 Communicating With Printers

This section describes how your program gains access to a printer and writes data to a printer. Specifically, this section covers the following topics:

- How to open a printer, write text to it, and pass control information to it using CONTROL operations and SETMODE functions. Accessing a Printer provides details.
- How to control laser printers and matrix line printers by sending escape sequences to them. Using the Printer Control Language provides an overview. Programming for Tandem Laser Printers and Programming for Tandem Matrix Line Printers provide details.
- How to recover from errors incurred while printing. See Recovering From Errors.

At the end of this section is a complete sample program that accesses a printer and responds to printer errors.

Most programs that send output to a printer do so indirectly by writing to a spooler collector. Some applications, however, need to write directly to the printer, especially if the user will need immediate notification of printer errors; for example, following a positioning error when printing paychecks.

Usually, you should write your programs to be able to write either to the spooler or directly to the printer. The purpose of writing to the spooler is to store for later printing the exact sequence of operations the program sent to the spooler. The only function lost by using the spooler is the ability to take special action if errors occur during printing.

For complete information about a specific printer, refer to the appropriate printer reference manual. For additional information about accessing printers, such as how to access a printer over a telephone line, refer to the appropriate data communications manual.

For complete programming details related to the spooler, see the spooler manuals.

Accessing a Printer

This subsection introduces the system procedures that relate to printer control and provides a skeleton program for printer access.

Procedures for Working With Printers

You access a printer the same way as you would any other file, by using file-system procedure calls. You use the following procedures to perform the indicated tasks with printers:

AWAITIO[X]	Checks for completion of a pending I/O operation. AWAITIO checks for completion of a READ, WRITE, or WRITEREAD operation. AWAITIOX checks for the completion of a READ, WRITE, WRITEREAD, READX, WRITEX, or WRITEREADX operation.
CANCEL	Cancels the oldest outstanding operation on an open printer.
CANCELREQ	Cancels a specified operation on an open printer.
CONTROL	Performs vertical forms-control functions.
DEVICE_GETINFOBYLDEV_	Provides the device type and configured record length of the device specified by logical device number as well as the CPU numbers where the primary and backup I/O processes run.
DEVICE_GETINFOBYNAME_	Provides the device type and configured record length of the device specified by name as well as the CPU numbers where the primary and backup I/O processes run.
FILE_CLOSE_	Stops access to an open printer.
FILE_GETINFO_	Provides error information and characteristics about an open printer.
FILE_OPEN_	Establishes communication with a printer.
SETMODE	Controls various printer functions.
SETMODENOWAIT	Does the same as SETMODE, except that printer functions are applied in a nowait manner.
WRITE[X]	Prints a line on the printer.

Table 11-1 summarizes all CONTROL operations that affect printer operation.

Table 11-1. Printer CONTROL Operations

CONTROL Number	Operation
1	Provides forms control
11	Specifies a wait for a modem connection
12	Disconnects a modem

On return from one of the calls listed in Table 11-1, the condition code should be CCE if the CONTROL operation was successful. A condition code of CCL indicates an error.

Table 11-2 summarizes all SETMODE functions that relate to printer operation.

Table 11-2. Printer SETMODE Functions

SETMODE Number	Function
5	Sets the system automatic perforation skip mode
6	Sets system spacing control
22	Sets the line printer baud rate
25	Sets the form length
26	Sets or clears vertical tabs
27	Sets system spacing mode
28	Resets configured values
29	Sets automatic answer mode or control answer mode
37	Gets the device status
68	Sets the horizontal pitch
260	Selects printer language (5577 only)

On return from one of the calls listed in Table 11-2, the condition code should be CCE if the SETMODE function was performed successfully. A condition code of CCL indicates an error. A condition code of CCG indicates that the attempted SETMODE function is invalid for the type of device.

For complete details of these procedure calls, CONTROL operations, and SETMODE functions, refer to the *Guardian Procedure Calls Reference Manual*.

A Printer Program Outline

The general approach to directly accessing a line printer from an application program is:

1. Open the printer by calling the FILE_OPEN_ procedure. Use the printer file name to identify the printer to the FILE_OPEN_ procedure. To prevent your printed messages being mixed with messages printed by other processes, you should open the printer for exclusive access.
2. For a matrix line printer, position the paper to the top of the form by using the CONTROL procedure. Operation 1 allows you to adjust the paper position.
3. Call the WRITE[X] procedure to print each line of text.
4. When you have finished using a matrix line printer, call the CONTROL procedure to position the paper again to the top of the form. Then call the FILE_CLOSE_ procedure to terminate your access to the printer.

The code fragments shown below illustrate this technique:

```
LITERAL MAXLEN = 256;
 .
 .
STRING   .PRINTER^NAME[0:MAXLEN - 1] := "$LP1";!printer
                                     ! name is $LP1
INT       PRINTER^NUM;               !printer file number
STRING   .SBUFFER[0:132];            !print buffer
STRING   .S^PTR;
LITERAL   EXCLUSIVE = ZSYS^VAL^OPENEXCL^EXCLUSIVE;
LITERAL   POSITION = 1;
LITERAL   TOP^OF^FORM = 0;
 .
 .
!Open the printer for exclusive access:
LENGTH := 4;
ERROR := FILE_OPEN_(PRINTER^NAME:LENGTH,
                    PRINTER^NUM,
                    EXCLUSIVE);
IF ERROR <> 0 THEN ...

!Move to the top of the form:
CALL CONTROL(PRINTER^NUM,
             POSITION,
             TOP^OF^FORM);
IF <> THEN ...
 .
 .
!Send text to printer:
SBUFFER ':=' "Print just one line on the printer" -> @S^PTR;
CALL WRITEX(SBUFFER,
            PRINTER^NUM,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

```
!Move to the top of the form:
CALL CONTROL(PRINTER^NUM,
             POSITION,
             TOP^OF^FORM);
IF <> THEN CALL DEBUG;
 .
 .
!Close the printer:
CALL FILE_CLOSE_(PRINTER^NUM);
 .
 .
```

Using the Printer Control Language

All HP printers support the printer control language (PCL). New printers introduced over the next few years will also support PCL.

PCL allows you to control the printer by sending escape sequences to it. The procedure-call interface to the file system also allows you to perform some of these escape sequences simply by calling the SETMODE or CONTROL procedure. The mapping of these calls to PCL escape sequences is done internally.

The functions provided by PCL vary depending on the type of printer you are using; for example, printers such as the 5577 and 5574 laser printers support a different subset of PCL than a matrix line printer such as the 5515. With a goal of printer compatibility, PCL has five levels of definition. Each printer type supports one of these levels:

- Level 1: **the print and space set** is a subset of commands for inexpensive printers that provides a simple way to produce hard copy.
- Level 2: **the EDP transaction feature set** supports multiple-user printers suitable for use in an EDP or transaction-oriented environment.
- Level 3: **the office word processing feature set** provides additional data-formatting capabilities.
- Level 4: **the page-formatting feature set** provides comprehensive formatting capabilities for the support of sophisticated printers such as laser printers.
- Level 5: **the enhanced page formatting feature set** provides additional formatting capabilities such as scalable outline fonts, reverse printing (white on black), and finer rotation increments.

For example, the 5577 and 5574 laser printers support PCL level 5; the 5573 and 5573D laser printers support level 4; the 5515/5516/5518 matrix line printers support level 2. Use care when writing programs that access printers to ensure that the feature set used is available on all the printers that you might want your program to work with.

This subsection describes some of the more common features of PCL. The following functions are among those supported by PCL:

- Job-control commands let you select the number of copies you want printed and whether you want duplexing.
- Page-control commands let you establish the page length and margins and provide forms control.
- Font-management commands allow you to select fonts, establish style and stroke weight, and so on.

For complete details of what PCL commands are available for the 5577 and 5574 laser printers, see the *PCL 5 Printer Language Technical Reference Manual*. For the 5573 and 5573D printers, refer to the *Tandem 5573 Laser LX Printer Reference Manual*. For details about Tandem matrix line printers, refer to the *Tandem 5515/5516/5518 Printers Technical Reference Manual*.

Controlling the Printer

You control any HP supported printer using escape sequences supported by PCL. For ease of use, some of these escape sequences have equivalent CONTROL operations or SETMODE functions. You therefore have three ways of sending control information to the printer: by issuing CONTROL procedure calls, by issuing SETMODE procedure calls, or by sending the escape sequence itself to the printer using the WRITE[X] procedure.

Controlling the Printer Using the CONTROL Procedure

The CONTROL procedure controls vertical positioning. For example, you use CONTROL operation 1 to position the paper at the top of the form. Vertical positioning is described later in this section.

Controlling the Printer Using the SETMODE Procedure

The SETMODE procedure performs functions such as resetting the printer and overstriking. These functions are described later in this section.

Controlling the Printer Using Escape Sequences

The WRITE[X] procedure sends escape sequences to the printer to perform any available printer function. Specifically, you send escape sequences to the printer for those operations for which there is no alternative CONTROL operation or SETMODE function. These functions include specifying print characteristics and underlining text.

An escape sequence is a series of characters that begins with the escape character (ASCII %33). Escape sequences are not printed, but they are interpreted by the printer.

Two types of escape sequences can be sent to Tandem printers: two-character escape sequences and parameterized escape sequences. Two-character escape sequences have the following general format:

Syntax for a two-character escape sequence:

`esc x`

`esc`	is the escape character (ASCII %33).
`x`	is an ASCII character that specifies the function the printer is to perform.

Parameterized escape sequences have the following general format:

Syntax for a parameterized escape sequence:

`esc param-char group-char parameters term-char`

`esc`	is the escape character (ASCII %33).
`param-char`	is an ASCII character—&, (, or)—that specifies that the escape sequence is a parameterized escape sequence.
`group-char`	is an ASCII character that specifies the type of function the printer is to perform.
`parameters`	is a string of ASCII characters. The meaning of these characters depends on the function specified by *`group-char`* and *`term-char`*.
`term-char`	is an ASCII character that specifies the precise function that the printer is to perform and marks the end of the parameters. This character can be uppercase or lowercase: 2 An uppercase character specifies the end of the escape sequence. 2 A lowercase character specifies that another escape sequence immediately follows. The *`esc`*, *`param-char`*, and *`group-char`* must be omitted from the escape sequence that follows this lowercase letter.

The following are examples of parameterized escape sequences:

```
esc&a99M
esc&a9L
```

The first example sets the right margin at character position 99. The second example sets the left margin at character position 9. These examples can be combined as follows:

```
esc&a99m9L
```

This escape sequence is equivalent to the previous two examples. Note that the example specifies a lowercase “m” rather than an uppercase “M.”

To send escape sequences to the printer, you must construct a string of ASCII characters according to the format of escape sequences given above and send those characters to the printer using the WRITE[X] procedure. This example sets the left and right margins.

```
SBUFFER ':=' [%33,"&a99m9L"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
```

Commonly Used PCL Escape Sequences

Table 11-3 lists some of the more commonly used PCL commands and indicates which Tandem printers support each command.

Table 11-3. Common PCL Escape Sequences (page 1 of 2)

Escape Sequence	Function Performed	5515/5516/5518 Matrix Line Printers	5574 and 5577 Laser Printers	5573 and 5573D Laser Printers
escE	Resets the printer	X	X	X
esc&a#L	Left margin	X	X	X
esc&a#M	Right margin	X	X	X
esc&a#P	Print direction (degrees in 90-degree increments)		X	
esc&d#D	Underline enable	X	X	X
esc&d@	Underline disable	X	X	X
esc&k#H	Horizontal motion index		X	X
esc&l#A	Paper size		X	X
esc&l#C	Vertical motion index		X	X
esc&l#D	Line spacing	X	X	X
esc&l#E	Top margin		X	X
esc&l#G	Output bin selection		X	
esc&l#H	Paper source		X	X
esc&l#O	Orientation		X	X
esc&l#P	Page length	X	X	X
esc&l#S	Simplex/duplex selection		X (5577 only)	
esc&l1T	Job separation		X	
esc&l#V	Select VFC channel	X		
esc&l#W	Programmable VFC	X		

Table 11-3. Common PCL Escape Sequences (page 2 of 2)

Escape Sequence	Function Performed	5515/5516/5518 Matrix Line Printers	5574 and 5577 Laser Printers	5573 and 5573D Laser Printers
esc&l#X	Number of copies		X	X
esc(ID	Primary symbol set	X	X	X
esc(s#B	Primary stroke weight		X	X
esc(s#H	Primary pitch	X	X	X
esc(s#P	Primary spacing	X	X	X
esc(s#Q	Primary font density	X		
esc(s#S	Primary style	X	X	X
esc(s#T	Primary typeface	X	X	X
esc(s#V	Primary height		X	X
esc(#X	Primary font selection by ID number		X	X
esc%-12345X	Universal exit language/start of PJL		X (5577 only)	

Programming for Tandem Laser Printers

This subsection describes some of the more commonly used programmable features of the supported Tandem laser printers. The supported laser printers include:

- 5573 laser printer supporting PCL 4
- 5573D laser printer supporting PCL 4
- 5574 laser printer supporting PCL 5
- 5577 laser printer supporting PCL 5 and PostScript

The information presented here describes how to use:

- Commands for selecting the printer language you want to use: PostScript or PCL
- Job-control commands to select the number of copies, simplex or duplex mode, the paper source, and the output bin, and to separate jobs
- Page-control commands to set the page size and length; the size of the left, right, and top margins; and the horizontal and vertical motion indexes
- Text-printing commands to select a font, the font size, and the orientation, and to underline text

For complete details of how to use all available PCL commands for the 5574 and 5577 laser printers, see the *PCL 5 Printer Language Technical Reference Manual*. For the

5573 and 5573D printers, refer to the *Tandem 5573 Laser LX Printer Reference Manual*.

Selecting a Printer Language (5577 Only)

The 5577 laser printer can accept commands written in PCL or PostScript printer language. You can select the language in one of two ways:

- Using SETMODE function 260
- Sending character sequences to the printer

Note. Language switching must be enabled by entering the SET SWITCH = ON command at the printer console panel if you intend to use either of the language-switching techniques. Without this command, the printer interprets switching commands as normal print data, which will either appear as printer output or cause unpredictable errors in the job output. See the *5577 Printer User's Reference Manual* for details about enabling language switching.

Using SETMODE 260 to Select the Printer Language

Call the SETMODE procedure specifying function 260 and use the *`param1`* parameter to select the printer language you require. Set *`param1`* to either 2 or 1 to select PostScript mode. If you set *`param1`* to 2, a system-generated carriage return is issued at the end of each line; if you set *`param1`* to 1, no system-generated carriage return is issued at the end of each line. If you specify either 2 or 1, the printer is returned to PCL mode at the end of the job. Set *`param1`* to 0 to select PCL 5.

The following example selects PostScript mode, with no system-generated carriage return issued at the end of each line:

```
CALL SETMODE(260,1);
```

Using Character Sequences to Select the Printer Language

Use a combination of the Universal Exit Language/Start PJL (printer job language) command and the @PJL enter-language command to change the printer language using character sequences.

1. Send a Universal Exit Language/Start PJL command at the beginning and end of each job. Doing so ensures proper language switching regardless of changes to the default language established at the printer console panel and establishes clear print-job boundaries.
2. Send the appropriate @PJL enter-language command.

Note. Do not follow the Universal Exit Language/Start PJL command with a carriage-return/line-feed sequence. At the start of a job, you must follow this command immediately with a @PJL command; otherwise, an implicit switch to the default language occurs. Similarly, at the end of a job, you should not follow the Universal Exit Language/Start PJL command with a carriage-return/line-feed sequence.

Use the following sequences to send the Universal Exit Language/Start PJL and @PJL enter-language commands:

Escape sequence to send the Universal Exit Language/Start PJL command:

`esc%-12345X`

Character sequence to enter PostScript mode:

`@PJL enter language = PostScript<LF>`

Character sequence to enter PCL mode:

`@PJL enter language = PCL<LF>`

The following example selects PostScript mode:

```
SBUFFER ':=' [%33,"%-12345X@PJL enter language = PostScript",
             %12] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Using Job-Control Commands

The Tandem laser printers support PCL commands that provide job-control capabilities such as selecting the number of copies and whether you want to print on one side of the paper or on both sides. The following paragraphs describe these features.

These features are supported on all Tandem laser printers except where noted.

Selecting the Number of Copies

You select how many copies of the print job you require by writing an escape sequence with the following format to the printer:

Escape sequence for specifying the number of copies to print:

`esc&l`*`copies`*`X`

`copies` indicates the number of copies of the job you want printed. The following example prints 5 copies of the current job:

```
SBUFFER ':=' [%33,"&l5X"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Selecting Simplex or Duplex Mode

If a job is printed in simplex mode, it is printed on one side of the paper. Jobs that print on both sides of the paper are duplex-mode jobs.

You select simplex or duplex mode by writing an escape sequence with the following format to the printer:

> Escape sequence for setting simplex/duplex mode:
>
> `esc&l`*`mode-number`*`S`

`mode-number` is 0 for simplex, 1 for duplex with long-edge binding, or 2 for duplex with short-edge binding. The default mode is simplex.

The following example sets duplex mode with long-edge binding:

```
SBUFFER ':=' [%33,"&l1S"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Note. Duplexing is supported only on the 5577 laser printer.

Selecting the Paper Source

The paper source designates one of two paper locations as the paper source for printing: the internal tray or manual input.

You select the paper source by writing an escape sequence with the following format to the printer:

> Escape sequence for selecting the paper source:
>
> `esc&l`*`tray-number`*`H`

For PCL 4, the options for *`tray-number`* are:

0	Print the current page without changing the paper source
1	Internal tray (the default source)
2	Manual paper feed

For PCL 5, the options are:

0	Print the current page without changing the paper source
1	Upper paper tray (the default source)
2	Manual paper feed

3 Manual envelope feed

4 Lower paper tray

6 Envelope feeder

The following example selects the manual paper feed as the paper source:

```
SBUFFER ':=' [%33,"&l2H"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Selecting the Output Bin

You select the output bin by writing an escape sequence with the following format to the printer:

Escape sequence for selecting the output bin:

`esc&l`*`bin-number`*`G`

`bin-number` is 1 for the upper bin and 2 for the lower bin.

The following example selects the lower bin:

```
SBUFFER ':=' [%33,"&l2G"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Note. Bin selection is supported only on laser printers that support PCL 5.

Separating Jobs

You issue the job separation sequence by writing an escape sequence with the following format to the printer:

Escape sequence for selecting the output bin:

`esc&l1T`

The following issues the job separation sequence:

```
SBUFFER ':=' [%33,"&l1T"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Note. The job separation feature is supported only on the PCL 5 laser printers.

Using Page-Control Commands

Page-control commands include a subset of escape sequences that allow you to control characteristics such as the size of the page, orientation, margins, and text spacing. This subsection presents some of the more common commands. Again, for complete details on all page-control commands, refer to the appropriate printer reference manual.

The features described here are supported on all Tandem laser printers.

Setting the Paper Size

You need to specify to Tandem laser printers the physical page size of the paper you intend to print on. Use an escape sequence with the following format:

Escape sequence to set the paper size:

`esc&l`*`size`*`A`

`size` indicates the paper size or envelope size as follows:

Paper sizes:

1	Executive (7.25 inch x 10.5 inch)
2	Letter (8.5 inch x 11 inch)
3	Legal (8.5 inch x 14 inch)
26	A4 (210 mm x 297 mm)

Envelope sizes:

80	Letter (Monarch 7.75) (3.875 x 7.5)
81	Business (Commercial 10) (4.125 x 9.5)
90	International DL (110 mm x 220 mm)
91	International C5 (162 mm x 229 mm)

The following example selects the legal page size:

```
SBUFFER ':=' [%33,"&l3A"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Setting the Logical Page Length

The logical page length on Tandem laser printers is controlled by issuing an escape sequence with the following format:

Escape sequence to set page size:

esc&l*lines*P

lines gives the maximum number of lines that each subsequent page can have. This is the size of the **logical page**. The logical page sets the bounds for future operations.

The following example sets the page size to 48 lines:

```
SBUFFER ':=' [%33,"&l48P"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Setting the Margins

To set the left, right, and top margins, you use the following escape sequences:

Escape sequence to set the left margin:

esc&a*column-number*L

Escape sequence to set the right margin:

esc&a*column-number*M

Escape sequence to set the top margin:

esc&l*line-number*E

The left and right margins are set according to the number of columns from the left or right edge, respectively, of the logical page. The top margin is set to the number of lines from the top of the logical page.

The following example sets the text area as 10 columns from the left and right edges of the logical page and five lines from the top of the logical page:

```
SBUFFER ':=' [%33,"&a10l10M", %33, "&l5E"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Setting the Horizontal and Vertical Motion Indexes

The horizontal motion index designates the distance between columns in 1/120-inch increments. Similarly, the vertical motion index designates the distance between rows in 1/48-inch increments. You set the horizontal and vertical motion indexes using the following escape sequences:

Escape sequence to set the horizontal motion index:

`esc&k`*`column-separation`*`H`

Escape sequence to set the vertical motion index:

`esc&l`*`row-separation`*`C`

Escape sequence to set the line spacing:

`esc&l`*`line-spacing`*`D`

Setting the line spacing has the same effect as setting the vertical motion index, but you specify the number of lines per inch instead of the distance between adjacent rows.

The following example sets the horizontal motion index to 14/120-inch and the vertical motion index to 5 lines per inch:

```
SBUFFER ':=' [%33,"&k14H", %33, "&l5D"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Printing Text

PCL supports several operations that affect the appearance of printed characters. Your program can do the following:

- Select a font and alter its characteristics
- Underline text

The following paragraphs describe how to use these features in an application program.

Selecting Font Characteristics

Several fonts are supplied with the printer; these fonts are referred to as internal fonts. You can add fonts to your printer by inserting font cartridges or downloading soft fonts.

For internal fonts and downloaded soft fonts, you can alter the font characteristics including:

- The typeface (Courier, Times Roman, and so on)
- The symbol set; for example, to correspond to a national standard
- The character spacing (fixed or proportional)
- The pitch (number of characters per horizontal inch—proportional fonts only)
- The point size or character height
- The style (upright or italic)
- The stroke weight or boldness
- The orientation (portrait or landscape)

You specify the font characteristics by writing escape sequences with the following formats to the printer:

Escape sequence for specifying the typeface for the primary font:

```
esc(sfont-numberT
```

Escape sequence for specifying the symbol set for the primary font:

```
esc(id
```

Escape sequence for specifying the spacing for the primary font:

```
esc(svalueP
```

Escape sequence for specifying the pitch for the primary font:

```
esc(spitch-valueH
```

Escape sequence for specifying the point size for the primary font:

```
esc(spoint-sizeV
```

Escape sequence for selecting the printing style for the primary font:

```
esc(sstyle-valueS
```

Escape sequence for selecting the stroke weight for the primary font:

```
esc(sdensity-valueB
```

Escape sequence for selecting orientation:

```
esc&lorientationO
```

Refer to the appropriate printer reference manual for a complete list of possible values for each of these escape sequences. The following example describes a font for the Courier typeface, with the ASCII symbol set, fixed spacing, 10 characters per inch, 12 point, upright, bold, in portrait orientation:

```
SBUFFER ':=' [%33,"(s3T",%33,"(0U",%33,"(s0P",%33,"(s10H",
              %33,"(s12V",%33,"(s0S",%33,"(s3B",
              %33,"&l0O"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Underlining Text

To underline text, write an escape sequence in the following format to the printer:

Escape sequence for underlining text:

esc&d*position*D

Escape sequence to turn off underlining:

esc&d@

All text following the underline escape sequence is printed underlined up to the escape sequence that turns off underlining. By default, underlining is turned off.

position specifies where the line is drawn with respect to the text. *position* can have two values:

0	Fixed position; always the same distance below the line of text
3	Floating position; depends on the underline distance of all fonts printed on the current line

The following code fragment shows an example for a Tandem laser printer. It uses floating position underlining:

```
STRING .START^UNDER[0:4] := [%33,"&d3D"];
STRING .STOP^UNDER[0:3] := [%33,"&d@"];
 .

!Send the start-underlining escape sequence:
CALL WRITEX(PRINTER^NUM,START^UNDER,$LEN(START^UNDER));
IF <> THEN ...

!Send the text to be printed underlined:
SBUFFER ':=' "This is underlined text," -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...

!Send the stop-underlining escape sequence:
CALL WRITEX(PRINTER^NUM,STOP^UNDER,STOP^UNDER);
IF <> THEN ...
```

```
!Subsequent text is not underlined:
SBUFFER ':=' " and this is not.  " -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

The above example prints the following text:

```
This is underlined text, and this is not.
```

Resetting the Laser Printer Default Values

You can reset the printer to its default values (those configured at the control panel) by sending an escape sequence in the following format to the printer:

Escape sequence to restore configured values:

`escE`

The following example shows how to do this in an application program.

```
STRING .RESET^PRINTER[0:1] := [%33,"E"];
 .
 .
CALL WRITEX(PRINTER^NUM,
            RESET^PRINTER,
            $LEN(RESET^PRINTER);
IF <> THEN ...
```

This escape sequence resets printer characteristics such as symbol set, pitch, and underlining.

Programming for Tandem Matrix Line Printers

This subsection describes some of the more commonly used programmable features of the Tandem 5515, 5516, and 5518 matrix line printers.

The information presented here describes how to use:

- Page-control commands to set the page length and the size of the left and right margins
- Forms-movement commands to vertically position the paper
- Text printing commands to set font characteristics, underline text, and perform overstriking

For complete details of how to use all available PCL commands for the 5515, 5516, and 5518 line matrix printers, see the *Tandem 5515/5516/5518 Printers Technical Reference Manual*.

Using Page-Control Commands

Page-control commands include a subset of escape sequences that allow you to control characteristics such as the length of the page and the left and right margins. This subsection presents some of the more commonly used commands. Again, for complete details on all page-control commands, refer to the printer reference manual.

Setting the Page Length

Page length on printers supported by Tandem is controlled by issuing an escape sequence with the following format:

Escape sequence to set page length:

`esc&l`*`lines`*`P`

`lines` gives the maximum number of lines that each subsequent page can have. This is the size of the **logical page**. The logical page sets the bounds for future operations.

The following example sets the page size to 48 lines:

```
SBUFFER ':=' [%33,"&l48P"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Setting the Margins

To set the left and right margins, you use the following escape sequences:

Escape sequence to set the left margin:

`esc&a`*`column-number`*`L`

Escape sequence to set the right margin:

`esc&a`*`column-number`*`M`

The left and right margins are set according to the number of columns from the left or right edge, respectively, of the logical page.

The following example sets the text area as 10 columns from the left and right edges of the logical page:

```
SBUFFER ':=' [%33,"&a10l10M"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Controlling Forms Movement

Vertical positioning is done on the 5515/5516/5518 printers by looking up values in the vertical form control (VFC) table in printer memory. You can do this in two ways:

- Use the CONTROL procedure with operation 1
- Use an escape sequence

Using CONTROL Operation 1 to Position the Paper

To vertically position the paper using CONTROL operation 1, you need to supply the function to be performed. This function is supplied as a parameter to CONTROL operation 1. The I/O process converts the parameter into an escape sequence that accesses the VFC table.

The following example positions the paper to the next one-half page:

```
LITERAL POSITION = 1,
        NEXT^HALF^PAGE = 5;
 .
 .
CALL CONTROL(PRINTER^NUM,
             POSITION,
             NEXT^HALF^PAGE);
IF <> THEN ...
```

Refer to the description of the CONTROL procedure in the *Guardian Procedure Calls Reference Manual* for a complete list of vertical positioning options for printers with subtype 7.

Using an Escape Sequence to Position the Paper

To position the paper using an escape sequence, you use an escape sequence with a format like this:

Escape sequence to position the paper:

`esc&l`*`channel-num`*`V`

`channel-num` indicates a channel number in the range 0 through 16 in the VFC table.

The VFC table contains one row for each line that can be printed on a page. Each row is made up of 17 columns called VFC channels.

Each row/column location in the VFC table contains either a 0 or a 1. A 0 indicates that the line **cannot** be accessed when the channel is selected. A 1 indicates that the line **can** be accessed when the channel is selected.

Table 11-4 shows part of the default VFC table. The printer software automatically calculates the default VFC table according to the number of lines on the logical page.

When you access the VFC table using an escape sequence (whether directly or indirectly using CONTROL operation 1), the printer advances to the next line that contains a 1 in the selected VFC channel. For example, if the printer is currently at line 5 and channel 6 is selected, the printer advances to line 10.

Note. When you are accessing the VFC table using CONTROL operation 1, the supplied parameter refers to the VFC channel number offset by one. Parameter 0 refers to channel 1, parameter 1 to channel 2, and so on. You cannot refer to channel 0 using a CONTROL call.

Refer to the *Tandem 5515/5516/5518 Printers Technical Reference Manual* for details on the default channel settings.

Table 11-4. Default VFC Table

Line Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Channel Number																				
Top of Physical page																				
Top of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bottom of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Single Spacing	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Double Spacing	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Triple Spacing	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
Half form	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Quarter form	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Tenth line	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Bottom of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Bottom of Form - 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Top of Form - 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Top of Form - 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Seventh line	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
Sixth line	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
Fifth line	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Fourth line	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0

Programming the VFC Table

You can program the VFC table so that each entry enables or disables the start of printing at a given line. You can change the values that are stored in channels 1 through 16. You cannot change the values stored in channel 0.

To change the VFC table, send an escape sequence to the printer in the following form:

Escape sequence for programming the VFC table:

`esc&l`*`byte-count`*`W`*`vfc-data`*

This escape sequence moves the binary data provided in *`vfc-data`* into the VFC table. The first word of *`vfc-data`* corresponds to row 0, channels 1 through 16; the second word corresponds to row 1, channels 1 through 16, and so on. *`byte-count`* indicates the number of bytes in *`vfc-data`*.

The following example sets up channel 13 to cause the paper to be positioned at the next eighth line (instead of seventh). All other channels remain unchanged.

```
STRUCT VF^DATA;
BEGIN
   STRING CONTROL^CHARS[0:5];
   INT    TABLE^DATA[0:16];
END;
 .
 .
VF^DATA.CONTROL^CHARS ':=' [%33,"&L9W"] -> @S^PTR;
VF^DATA.TABLE^DATA ':=' [
              %B1011111100011111,    !line 0
              %B0010000000000000,    !line 1
              %B0011000000000000,    !line 2
              %B0010100000000000,    !line 3
              %B0011001000000001,    !line 4
              %B0010000000000010,    !line 5
              %B0011100000000100,    !line 6
              %B0010000000000000,    !line 7, channel 13 off
              %B0011011000001001,    !line 8, channel 13 on
              %B0010100000000000,    !line 9
              %B0011000100000010,    !line 10
              %B0010000000000000,    !line 11
              %B0011101000000101,    !line 12
              %B0010000001000000,    !line 13
              %B0111000010000000,    !line 14, channel 13 off
              %B0000000000000000,    !line 15
              %B0000000000001000,    !line 16, channel 13 on
                 ];
CALL WRITEX(PRINTER^NUM,VFC^DATA,$LEN(VF^DATA));
IF <> THEN ...
```

The VFC table now contains the values shown in Table 11-4.

Table 11-5. Modified VFC Table

Line Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Channel Number																				
Top of Physical page																				
Top of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bottom of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Single Spacing	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Double Spacing	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Triple Spacing	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
Half form	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Quarter form	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Tenth line	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Bottom of Form	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Bottom of Form - 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Top of Form - 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Top of Form - 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Eighth line	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
Sixth line	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0
Fifth line	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Fourth line	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0

Figure 11-1. Modified VFC Table

Channel Number	Top of Physical Page	Top of Form	Bottom of Form	Single Spacing	Double Spacing	Triple Spacing	Half Form	Quarter Form	Tenth Line	Bottom of Form	Bottom of Form—1	Top of Form —1	Top of Form —1	**Eighth Line**	Sixth Line	Fifth Line	Fourth Line
Line Number	0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16
0		0	0	1	1	1	1	1	1	0	0	0	1	**1**	1	1	1
1		0	0	1	0	0	0	0	0	0	0	0	0	**0**	0	0	0
2		0	0	1	1	0	0	0	0	0	0	0	0	**0**	0	0	0
3		0	0	1	0	1	0	0	0	0	0	0	0	**0**	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	**0**	0	0	1
5		0	0	1	0	0	0	1	0	0	0	0	0	**0**	0	1	0
6		0	0	1	1	1	0	0	0	0	0	0	0	**0**	1	0	0
7		0	0	1	0	0	0	0	0	0	0	0	0	**0**	0	0	0
8		0	0	1	1	0	0	0	0	0	0	0	0	**1**	0	0	1
9		0	0	1	0	1	0	0	0	0	0	0	0	**0**	0	0	0
10		0	0	1	1	0	1	1	1	0	0	0	0	**0**	0	1	0
11		0	0	1	0	0	0	0	0	0	0	0	0	**0**	0	0	0
12		0	0	1	1	1	0	0	0	0	0	0	0	**0**	1	0	1
13		0	0	1	0	0	0	0	0	0	0	0	0	**0**	0	0	0
14		0	0	1	1	0	0	0	0	0	0	0	0	**0**	0	0	0
15		0	0	1	0	1	0	1	0	0	0	0	0	**0**	0	1	0
16		0	0	1	1	0	0	0	0	0	0	0	0	**1**	0	0	1
17		0	0	1	0	0	0	0	0	0	0	0	0	**0**	0	0	0
18		0	0	1	1	1	0	0	0	0	1	0	0	**0**	1	0	0
19		0	1	1	0	0	0	0	0	1	0	1	0	**0**	0	0	0

VST120.VSD

Printing Text

PCL supports several operations that affect the appearance of printed characters on Tandem matrix line printers. Your program can specify the following:

- Font characteristics, including:
  - A symbol set; for example, to correspond to a national standard
  - The pitch (number of characters per horizontal inch)
  - The style (upright or italic)
  - The density or boldness (number of dots per character)
- Text underlining
- Text overstriking

The following paragraphs describe how to use these features in an application program.

Selecting Font Characteristics

You specify the font characteristics by writing escape sequences with the following formats to the printer:

Escape sequence for specifying the symbol set for the primary font:

`esc(`*`id`*

Escape sequence for specifying the pitch for the primary font:

`esc(s`*`pitch-value`*`H`

Escape sequence for selecting the printing style for the primary font:

`esc(s`*`style-value`*`S`

Escape sequence for selecting the print density for the primary font:

`esc(s`*`density-value`*`Q`

Refer to the appropriate printer reference manual for a complete list of possible values for each of these escape sequences. The following example describes a font for the Japanese ASCII symbol set, a pitch of 12 characters per inch, italic, with standard density:

```
SBUFFER ':=' [%33,"(0K",%33,"(s12H",%33,"(s1S",
               %33,"(s0Q"] -> @S^PTR;
CALL WRITEX(PRINTER^NUM,
              SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Note. You cannot change the pitch in the middle of a line.

Underlining Text

To underline text, write an escape sequence in the following format to the printer:

Escape sequence for underlining text:

`esc&dD`

Escape sequence to turn off underlining:

`esc&d@`

All text following the underline escape sequence is printed underlined up to the escape sequence that turns off underlining. By default, underlining is turned off.

The following code fragment shows an example for the 5515/5516/5518 printers:

```
STRING .START^UNDER[0:3] := [%33,"&dD"];
STRING .STOP^UNDER[0:3] := [%33,"&d@"];
 .
!Send the start-underlining escape sequence:
CALL WRITEX(PRINTER^NUM,START^UNDER,$LEN(START^UNDER));
IF <> THEN ...

!Send the text to be printed underlined:
SBUFFER ':=' "This is underlined text," -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...

!Send the stop-underlining escape sequence:
CALL WRITEX(PRINTER^NUM,STOP^UNDER,$LEN(STOP^UNDER));
IF <> THEN ...

!Subsequent text is not underlined:
SBUFFER ':=' " and this is not.  " -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
```

The above example prints the following text:

```
This is underlined text, and this is not.
```

Overstriking Text

The 5515/5516/5518 printers support two ways to print characters on top of each other:

- Print adjacent lines on top of each other by calling the SETMODE procedure with function 6 selected.

- Print groups of adjacent characters on top of each other by inserting backspace control codes.

Using SETMODE 6 to Overstrike Characters

You can use SETMODE function 6 to print a line on top of another line. SETMODE function 6 controls the spacing after you write a line to the printer. By default, a line feed and carriage return are performed after each line is printed. By calling SETMODE function 6, you can suppress the line feed.

The method for overstriking lines using SETMODE function 6 is as follows:

1. Call SETMODE function 6 specifying suppression of the line feed after printing a line. (Note that you do this **before** printing the line that you want to overstrike.)
2. Call the WRITE procedure to print the first line.
3. Call SETMODE function 6 to turn off suppression of the line feed.
4. Call the WRITE procedure to print the second line. The second line is printed on top of the first.

The following example shows the use of SETMODE function 6 to overstrike a line of text:

```
LITERAL SPACE^MODE = 6, NO^LF = 0, LF = 1;
 .
 .
!Suppress the line feed:
CALL SETMODE(PRINTER^NUM,SPACE^MODE,NO^LF);
IF <> THEN ...

SBUFFER ':=' "Denote blanks by b.  " -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...

!Turn on automatic line feed:
CALL SETMODE(PRINTER^NUM,SPACE^MODE,LF);
IF <> THEN ...

!Print the second line:
SBUFFER ':=' "                 /  " -> @S^PTR;
CALL WRITEX(PRINTER^NUM,SBUFFER,@S^PTR '-' @SBUFFER);
IF <> THEN ...
 .
 .
```

The “/” character in the second line overstrikes the “b” character of the first line.

Using Backspace Control Codes to Overstrike Characters

To overstrike a single character or small group of characters in a line, it might be easier to insert the backspace control code into the output buffer. The backspace control code has the ASCII value %10.

The following example shows the use of the backspace control code to overstrike one character:

```
SBUFFER ':=' ["Denote blanks by b",%10,"/.  "] -> @S^PTR;
CALL WRITEX(PRINTER^NUMBER,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
IF <> THEN ...
```

Again, the “/” character overstrikes the “b.”

Resetting the Printer to Default Values

You can reset the printer to its default values (those configured at the control panel) by sending an escape sequence in the following format to the printer:

Escape sequence to restore configured values:

`escE`

The following example shows how to do this in an application program.

```
STRING .RESET^PRINTER[0:1] := [%33,"E"];
 .
 .
CALL WRITEX(PRINTER^NUM,
            RESET^PRINTER,
            $LEN(RESET^PRINTER));
IF <> THEN ...
```

This escape sequence resets printer characteristics such as symbol set, pitch, and underlining. For the 5515/5516/5518 printers, a new default VFC table is calculated.

Recovering From Errors

The following errors require special consideration for all line printers:

100	Device not ready
102	Device out of paper
200-255	Path errors

When dealing with these errors, you should also consider whether your program is using nowait I/O and if so, whether multiple I/O operations are allowed concurrently. If your program does permit multiple concurrent I/O operations, lines may be missing or appear printed out of order.

Recovering From a “Device Not Ready” Error

Your application must be able to handle a “not ready” or “paper out” condition. With some printers, either condition causes a “device not ready” error (refer to the printer manual). If either of these conditions arises, your program should send a message to the user or system operator. Your application should then wait for the user to respond, indicating that the printer is ready.

The FILEERROR procedure is useful with devices that might generate retryable errors. You should call this procedure after checking the condition code following an I/O operation with the printer. FILEERROR returns a status value of 1 if the operation should be retried or 0 if it should not be retried. For errors that need to be retried, FILEERROR responds as follows:

- For error 100 or error 102, FILEERROR displays an appropriate message on the home terminal and waits for a reply. The user is then expected to fix the problem before typing a reply. To continue, the user presses the return key; FILEERROR returns 1. To discontinue, the user enters STOP; FILEERROR returns 0.
- For path errors (errors 200 through 255) FILEERROR returns 1 where it is appropriate for your program to retry the operation. For errors 200 and 201, FILEERROR returns a 1 if it can establish a path to the file in error; otherwise, it returns 0. For errors 240 and 241, FILEERRORS always returns a 1. For other path errors, FILEERRORS always returns a 0.

The following example shows one way of using the FILEERROR procedure:

```
ERROR := 1;
WHILE ERROR DO
BEGIN
   CALL WRITEX(PRINTERNUM,SBUFFER,WCOUNT);
   IF <> THEN
   BEGIN
      IF NOT FILEERROR(PRINTERNUM) THEN
         CALL PROCESS_STOP_(!process^handle!,
                            !specifier!,
                            ABEND);
   END;
   ELSE ERROR := 0;
END;
```

Recovering From Path Errors

Path-error recovery on a printer requires some special considerations because of paper movement. If a path error is detected and it is either error 200 or 201, the operation never got started. These operations can be retried if one of these errors occurs.

If a path error is detected and it is one of errors 210 through 231, the operation failed at some indeterminate point and paper movement may have occurred. Depending on the application, different approaches to error recovery are required. If the operation is critical, such as printing payroll checks, the check should be canceled and a message sent to the operator. However, if the information being printed is not considered critical, the line can be reprinted (and may thus be duplicated).

Sample Program for Using a Printer

The following example modifies the inventory program developed in Section 5, Communicating With Disk Files. It now includes an option to print the contents of the data file. This example also includes logic to read the name of the printer or spooler collector you wish to print to from the OUT file named in the Startup message.

The example adds some new procedures and makes changes to the MAIN and GET^COMMAND procedures as follows:

- The MAIN and GET^COMMAND procedures now contain logic to process an option “p” to print the contents of the data file.
- Option “p” selects the PRINT^FILE procedure. This procedure puts one print line of information in a buffer and then sends the buffer to the PRINT^OUT procedure for printing. PRINT^FILE does this by reading each record in turn from the data file, taking each field in turn, putting the information into the buffer with suitable leading text, and then calling PRINT^OUT.
- In addition to sending the formatted buffer to the line printer, the PRINT^OUT procedure also performs error checking and error processing. By calling the FILEERROR procedure, it is able to decide whether to retry a particular error, wait for a user response before retrying, or abend the operation.
- The INIT and SAVE^STARTUP^MESSAGE procedures have been added to perform file initialization. The terminal file name is taken from the IN file name of the Startup message.

Note. This example assumes a 5515/5516/5518 printer. To make the example work with another type of printer, you need to change the escape sequences used by the PRINT^FILE procedure for underlining text for the mechanism used on the printer you have.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;  !maximum file-
                                          ! name length
LITERAL OLD     = 0;
LITERAL NEW     = 1;
LITERAL BUFSIZE = 132;
LITERAL PARTSIZE= 6;
LITERAL DESCSIZE= 60;
LITERAL SUPPSIZE= 60;
LITERAL ABEND   = 1;

STRING .SBUFFER[0:BUFSIZE];       !I/O buffer (one extra char)
STRING .S^PTR;                    !pointer to end of string
INT     PARTFILE^NUM;             !part file number
INT     TERMNUM;                 !terminal file number

STRUCT .PART^RECORD;             !database record
BEGIN
   STRING PART^NUMBER[0:5];
   STRING DESCRIPTION[0:59];
   INT    DESC^LEN;
   STRING SUPPLIER[0:59];
   INT    SUP^LEN;
   INT    ON^HAND;
   INT    UNIT^PRICE;
END;

STRUCT CI^STARTUP;               !Startup message
BEGIN
   INT    MSGCODE;
   STRUCT DEFAULTS;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
END;

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (INITIALIZER,
?  FILE_OPEN_,WRITEREADX,WRITEX,KEYPOSITION,NUMIN,
?  PROCESS_STOP_,READX,POSITION,DNUMOUT,FILE_GETINFO_,
?  READUPDATEX,WRITEUPDATEX,DNUMIN,READUPDATELOCKX,
?  WRITEUPDATEUNLOCKX,FILEERROR,CONTROL,FILE_CLOSE_,
?  OLDFILENAME_TO_FILENAME_,UNLOCKREC)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to format
! and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =     @S^PTR := @SBUFFER #;


!  Put a string into the line:

   DEFINE  PUT^STR (S) =    S^PTR ':=' S -> @S^PTR #;


!  Put an integer into the line:

   DEFINE  PUT^INT (N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;


!  Print a line:

   DEFINE  PRINT^LINE  =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;


!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;


!  Print a string:

   DEFINE  PRINT^STR (S) =  BEGIN   START^LINE;
                                    PUT^STR(S);
                                    PRINT^LINE;  END   #;
```

```
!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when the
! file is not open when there is no file number for it.
! FILE^ERRORS is used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING    .FNAME;
INT        LEN;
INT        ERROR;
BEGIN

!  Compose and print the message

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_(!process^handle!,
                      !specifier!,
                      ABEND);
END;

!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!-------------------------------------------------------------

PROC FILE^ERRORS (FNUM);
INT         FNUM;
BEGIN
   INT      ERROR;
   STRING  .FNAME[0:MAXFLEN - 1];
   INT      FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! This procedure writes a message on the terminal and checks
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING  .BUF;
INT      LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);
END;

!------------------------------------------------------------
! This procedure asks the user for the next function to do:
!
! "r" to read records
! "u" to update a record
! "i" to insert a record
! "p" to print records
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' to Read Record, ");
   PRINT^STR("     'u' to Update a Record,  ");
   PRINT^STR("     'i' to Insert a Record, ");
   PRINT^STR("     'p' to Print Records, ");
   PRINT^STR("     'x' to Exit.  ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER, @S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to display a part record on the terminal
!------------------------------------------------------------

PROC DISPLAY^RECORD;
BEGIN

   PRINT^BLANK;

!  Display part number:

   PRINT^STR("Part Number Is:    " & PART^RECORD.PART^NUMBER
                                               FOR PARTSIZE);

!  Display part description:

   PRINT^STR("Part Description:  " & PART^RECORD.DESCRIPTION
                                   FOR PART^RECORD.DESC^LEN);

!  Display part supplier name:

   PRINT^STR("Supplier:          " & PART^RECORD.SUPPLIER
                                   FOR PART^RECORD.SUP^LEN);

!  Display quantity on hand:

   START^LINE;
   PUT^STR("Quantity on hand:  ");
   PUT^INT(PART^RECORD.ON^HAND);
   PRINT^LINE;

!  Display unit price:

   START^LINE;
   PUT^STR("Unit Price:       $");
   PUT^INT(PART^RECORD.UNIT^PRICE);
   PRINT^LINE;
END;
```

```
!------------------------------------------------------------
! Procedure to prompt user for input to build a new record.
!------------------------------------------------------------

PROC ENTER^RECORD(TYPE);
INT  TYPE;

BEGIN
   INT     COUNT^READ;
   INT     STATUS;
   STRING .NEXT^ADDR;

   DEFINE BLANK^FILL(F) =
          F ':=' " " & F FOR $LEN(F)*$OCCURS(F) - 1 BYTES #;

   PRINT^BLANK;

!  If inserting a new record, prompt for a part number.
!  If updating an exiting record, record number is already
!  known:

   IF TYPE = NEW THEN
   BEGIN
      SBUFFER ':=' "Enter Part Number:  " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      BLANK^FILL(PART^RECORD.PART^NUMBER);
      PART^RECORD.PART^NUMBER ':=' SBUFFER FOR
                                   $MIN(COUNT^READ,PARTSIZE);
   END;

!  Prompt for a part description:

   SBUFFER ':=' "Enter Part Description:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,DESCSIZE);
      BLANK^FILL(PART^RECORD.DESCRIPTION);
      PART^RECORD.DESCRIPTION ':=' SBUFFER FOR COUNT^READ;
      PART^RECORD.DESC^LEN := COUNT^READ;
   END;
```

```
!  Prompt for the name of the supplier:

   SBUFFER ':=' "Enter Supplier Name:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      COUNT^READ := $MIN(COUNT^READ,SUPPSIZE);
      BLANK^FILL(PART^RECORD.SUPPLIER);
      PART^RECORD.SUPPLIER ':=' SBUFFER FOR COUNT^READ;
      PART^RECORD.SUP^LEN := COUNT^READ;
   END;

!  Prompt for the quantity on hand:

PROMPT^AGAIN:
   SBUFFER ':=' "Enter Quantity On Hand:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR := NUMIN(SBUFFER,PART^RECORD.ON^HAND,10,
                          STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN;
      END;
   END;

!  Prompt or unit price:

PROMPT^AGAIN1:
   SBUFFER ':=' "Enter Unit Price:  $" -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   IF TYPE = NEW OR COUNT^READ > 0 THEN
   BEGIN
      SBUFFER[COUNT^READ] := 0;
      @NEXT^ADDR := NUMIN(SBUFFER,PART^RECORD.UNIT^PRICE,10,
                          STATUS);
      IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ] THEN
      BEGIN
         PRINT^STR("Invalid number");
         GOTO PROMPT^AGAIN1;
      END;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to stop printing.  This procedure positions the
! paper at the top of the form and closes the printer.
!-------------------------------------------------------------

PROC FORMFEED^AND^CLOSE(PNUM);
INT PNUM;
BEGIN
   LITERAL POSITION = 1;
   LITERAL TOP^OF^FORM = 0;

!  Position the paper to the top-of-form:

   CALL CONTROL(PNUM,POSITION,TOP^OF^FORM);
   IF <> THEN CALL FILE^ERRORS(PNUM);

!  Close the printer:

   CALL FILE_CLOSE_(PNUM);
   IF <> THEN CALL FILE^ERRORS(PNUM);

END;


!-------------------------------------------------------------
! Procedure for printing a line on the printer.  This
! procedure returns when the line has been successfully
! printed.
!
! If printing is unsuccessful, then the FILEERROR procedure
! offers the user the option of trying again.
!-------------------------------------------------------------

PROC PRINT^OUT(PRINTER^NUM,SBUFFER,WCOUNT);
INT     PRINTER^NUM;
STRING .SBUFFER;
INT     WCOUNT;

BEGIN
   INT ERROR;

   ERROR := 1;
   WHILE ERROR DO
   BEGIN
      CALL WRITEX(PRINTER^NUM,SBUFFER,WCOUNT);
      IF <> THEN
      BEGIN
         IF NOT FILEERROR(PRINTER^NUM)
            THEN CALL FILE^ERRORS(PRINTER^NUM);
      END
      ELSE ERROR := 0;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure for printing records.  The user selected "p."
! The procedure prints out the entire file, six records
! to a page.
!------------------------------------------------------------

PROC PRINT^FILE;
BEGIN
   STRING .PRINTER^NAME[0:MAXFLEN];
   INT     PRINTERNUM;
   INT     PLEN;
   INT     ERROR;
   LITERAL EXCLUSIVE = 1;
   LITERAL POSITION = 1;
   LITERAL TOP^OF^FORM = 0;

!  Open the printer with exclusive access, using the OUT file
!  from the Startup message:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.OUTFILE.VOLUME,
                        PRINTER^NAME:MAXFLEN,
                        PLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                      !specifier!,
                                      ABEND);
   ERROR := FILE_OPEN_(PRINTER^NAME:PLEN,PRINTERNUM,
                    !access!,
                    EXCLUSIVE);
   IF ERROR <> 0
      THEN CALL FILE^ERRORS^NAME(PRINTER^NAME:PLEN,ERROR);

!  Position paper to top of form:

   CALL CONTROL(PRINTERNUM,POSITION,TOP^OF^FORM);
   IF <> THEN CALL FILE^ERRORS(PRINTERNUM);

!  Position to the start of the parts file:

   SBUFFER ':=' "0";
   CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,
                 !key^specifier!,
                 1,0);
   IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

!  Loop until all records printed:

   WHILE 1 DO
   BEGIN
```

```
!  Read a record.  Return to PARTS if end of file:

   CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
      IF ERROR = 1 THEN
      BEGIN
         CALL FORMFEED^AND^CLOSE(PRINTERNUM);
         RETURN;
      END;
      CALL FILE^ERRORS(PARTFILE^NUM);
   END;

!  Print the part number:

   START^LINE;
   S^PTR ':=' [%33,"&dDPart Number Is:",%33,"&d@    "]
              -> @S^PTR;
   S^PTR ':=' PART^RECORD.PART^NUMBER FOR 6 -> @S^PTR;
   CALL PRINT^OUT(PRINTERNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Print the part description:

   START^LINE;
   S^PTR ':=' [%33,"&dDPart Description:",%33,"&d@  "]
              -> @S^PTR;
   S^PTR ':=' PART^RECORD.DESCRIPTION FOR
              PART^RECORD.DESC^LEN -> @S^PTR;
   CALL PRINT^OUT(PRINTERNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Print the part supplier name:

   START^LINE;
   S^PTR ':=' [%33,"&dDSupplier:",%33,"&d@          "]
              -> @S^PTR;
   S^PTR ':=' PART^RECORD.SUPPLIER FOR PART^RECORD.SUP^LEN
              -> @S^PTR;
   CALL PRINT^OUT(PRINTERNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Print the quantity on hand:

   START^LINE;
   S^PTR ':=' [%33,"&dDQuantity on hand:",%33,"&d@  "]
              -> @S^PTR;
   PUT^INT(PART^RECORD.ON^HAND);
   CALL PRINT^OUT(PRINTERNUM,SBUFFER,@S^PTR '-' @SBUFFER);
   !  Print the unit price:

      START^LINE;
      S^PTR ':=' [%33,"&dDUnit Price:",%33,"&d@        $"]
                 -> @S^PTR;
      PUT^INT(PART^RECORD.UNIT^PRICE);
      CALL PRINT^OUT(PRINTERNUM,SBUFFER,@S^PTR '-' @SBUFFER);

      CALL PRINT^OUT(PRINTERNUM,SBUFFER,0);
   END;

END;

!------------------------------------------------------------
! Procedure for reading records.  The user selected function
! "r."  The start of the read is selected by approximate key
! positioning.  The user has the option of sequentially
! reading subsequent records.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT         COUNT^READ;
   INT         ERROR;

!  Prompt the user for the part number:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);


!  Position approximately to the selected record:

   CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,
                    !key^specifier!,
                    COUNT^READ,0);
   IF <> THEN  CALL FILE^ERRORS(PARTFILE^NUM);
```

```
!  Loop reading and displaying records until user declines
!  to read the next record (any response other than "y"):

   DO BEGIN

      PRINT^BLANK;

      !  Read a record from the part file.
      !  If end-of-file is reached,
      !  return control to the main procedure.

      CALL READX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record");
            RETURN;
         END;
         CALL FILE^ERRORS(PARTFILE^NUM);
      END;

   !  Display the record on the terminal:

      CALL DISPLAY^RECORD;

      PRINT^BLANK;

      !  Prompt the user to read the next record (user must
      !  respond "y" to accept, otherwise return to select
      !  next function):

      SBUFFER ':=' ["Do you want to read another ",
                                 "record (y/n)? "]
                     -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN  CALL FILE^ERRORS(TERMNUM);

      SBUFFER[COUNT^READ] := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

```
!------------------------------------------------------------
! Procedure for updating a record.  The user selected
! function "u."  The user is prompted to enter the part
! number of the record to be updated, then the old contents
! are displayed on the user's terminal before prompting the
! user to enter the updated record.
!------------------------------------------------------------

PROC UPDATE^RECORD;
BEGIN

   INT     COUNT^READ;
   INT     ERROR;
   STRUCT .SAVE^REC(PART^RECORD);
   STRUCT .CHECK^REC(PART^RECORD);

   PRINT^BLANK;

!  Prompt the user for the part number of the record to be
   updated:

   PRINT^BLANK;
   SBUFFER ':=' "Enter Part Number:  "  -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN  CALL FILE^ERRORS(TERMNUM);


!  Position exactly to the selected record.  First pad the
!  key with blanks in case the full length was not entered:

   SBUFFER[COUNT^READ] ':=' [PARTSIZE * [" "]];
   CALL KEYPOSITION(PARTFILE^NUM,SBUFFER,
                    !key^specifier!,
                    !length^word!,
                    2);
   IF <> THEN  CALL FILE^ERRORS(PARTFILE^NUM);

!  Read the selected record.  If no such record exists,
!  the procedure informs the user and returns control to
!  the main procedure:

   CALL READUPDATEX(PARTFILE^NUM,PART^RECORD,
                        $LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
      IF ERROR = 11 THEN
```

```
         BEGIN
            PRINT^BLANK;
            START^LINE;
            PUT^STR("No such record");
            PRINT^LINE;
            RETURN;
         END
         ELSE CALL FILE^ERRORS(PARTFILE^NUM);
      END;

   !  Save the record for later comparison:

      SAVE^REC ':=' PART^RECORD FOR $LEN(PART^RECORD) BYTES;

   !  Display the record on the terminal:

      CALL DISPLAY^RECORD;

   !  Prompt the user for the updated record:

      CALL ENTER^RECORD(OLD);

   !  Now that the user has entered the changes, reread the
   !  record and check to see whether someone else changed it
   !  while the user was responding:

      CALL READUPDATELOCKX(PARTFILE^NUM,CHECK^REC,
                           $LEN(PART^RECORD));
      IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);

      IF CHECK^REC <> SAVE^REC FOR $LEN(PART^RECORD) BYTES THEN
      BEGIN
         CALL UNLOCKREC(PARTFILE^NUM);
         PRINT^BLANK;
         PRINT^STR("The record was changed by someone else " &
                   "while you were working on it.");
         PRINT^STR("Your change was not made.");
         RETURN;
      END;

   !  Write the new record to the file:

      CALL WRITEUPDATEUNLOCKX(PARTFILE^NUM,PART^RECORD,
                              $LEN(PART^RECORD));
      IF <> THEN CALL FILE^ERRORS(PARTFILE^NUM);
   END;
```

```
!-------------------------------------------------------------
! Procedure for inserting a record.  The user selected
! function "i." The user is prompted to enter the new record.
! The procedure inserts the new record in the appropriate
! place in the file.
!-------------------------------------------------------------

PROC INSERT^RECORD;
BEGIN

   INT     ERROR;

   PRINT^BLANK;

!  Prompt the user for the new record:

   CALL ENTER^RECORD(NEW);

!  Write the new record to the file:

   CALL WRITEX(PARTFILE^NUM,PART^RECORD,$LEN(PART^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(PARTFILE^NUM,ERROR);
      IF ERROR = 10 THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("A record exists with that part number.");
         PRINT^STR("Your new one was not entered.");
      END
      ELSE
      BEGIN
         CALL FILE^ERRORS(PARTFILE^NUM);
      END;
   END;
END;


!-------------------------------------------------------------
! Procedure to exit the program.
!-------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure to process an illegal command.  The procedure
! informs the user that the selection was other than "r,"
! "u," "i," "p," or "x."
!------------------------------------------------------------

PROC ILLEGAL^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that his selection was invalid
!  then return to prompt again for a valid function:

   PRINT^STR("ILLEGAL COMMAND: "  &
             "Type either 'r,' 'u,' 'i,' 'p,' or 'x'");
END;


!------------------------------------------------------------
! Procedure to save the Startup message in the CI^STARTUP
! global structure.
!------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH,MATCH)VARIABLE;
INT .RUCB;
INT .START^DATA;
INT .MESSAGE;
INT  LENGTH;
INT  MATCH;
BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;
```

```
!------------------------------------------------------------
! This procedure does the initialization for the program.
! It calls INITIALIZER to dispose of the startup messages.
! It opens the home terminal and the data file used by the
! program.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .PARTFILE^NAME[0:MAXFLEN - 1];   !name of part file
   INT     PARTFILE^LEN;
   STRING .TERM^NAME[0:MAXFLEN - 1];       !terminal file
   INT     TERMLEN;
   INT     ERROR;

!  Read and save startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the terminal file (the IN file):

   ERROR := OLDFILENAME_TO_FILENAME_(
                           CI^STARTUP.INFILE.VOLUME,
                           TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERMNUM);
   IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                 !specifier!,
                                 ABEND);

!  Open the part file with a sync depth of 1:

   PARTFILE^NAME ':=' "$XCEED.DJCEGD10.PARTFILE" -> @S^PTR;
   PARTFILE^LEN := @S^PTR '-' @PARTFILE^NAME;
   ERROR := FILE_OPEN_(PARTFILE^NAME:PARTFILE^LEN,
                       PARTFILE^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       1);
   IF <> THEN
      CALL FILE^ERRORS^NAME(PARTFILE^NAME:PARTFILE^LEN,ERROR);
END;
```

```
!------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize, then it goes into a loop calling GET^COMMAND to
! get the next user request and calling the procedure to
! carry out that request.
!------------------------------------------------------------

PROC PARTS MAIN;
BEGIN
   STRING  CMD;

   CALL INIT;

!  Loop indefinitely until user selects function x:

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command:

      CMD := GET^COMMAND;

   !  Call the function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R"  -> CALL READ^RECORD;

         "u", "U"  -> CALL UPDATE^RECORD;

         "i", "I"  -> CALL INSERT^RECORD;

         "p", "P"  -> CALL PRINT^FILE;

         "x", "X"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL ILLEGAL^COMMAND;
      END;
   END;
END;
```

12

Communicating With Magnetic Tape

Magnetic tapes used on a HP system can be labeled or unlabeled. A labeled tape contains standard ANSI, IBM, BACKUP, or TMF tape labels that identify files and tape volumes and control access. Any tape that has none of these tape labels is an unlabeled tape; it is up to the application to interpret any file or volume header information or to work without this information.

Labeled tape support provides a mechanism for accessing tapes produced by other vendors and a way to create tapes on a system to be read by another vendor's system. In addition, labeled tapes provide a convenient way for applications on HP systems to maintain databases on magnetic tape.

This section discusses the programmatic interface with magnetic tapes, describing the following topics in this order:

- Basic magnetic tape operations that are mostly common to labeled and unlabeled tapes. These operations include how to position the tape by record or file, how to read and write tape records, how to block records for efficiency, and how to further improve performance by using buffered mode.
- Operations specific to labeled tapes, including how to open a labeled tape file and how to set the attributes for a file using DEFINEs. Examples are included on how to set attributes for reading and writing single-file labeled tapes, multiple-file labeled tapes, and files that occupy multiple labeled tapes.
- A complete sample program to maintain data on labeled tape.
- Operations specific to unlabeled tapes. These include how to open an unlabeled tape file as well as some guidelines on how to access single-file unlabeled tapes, multiple-file unlabeled tapes, and files contained on more than one unlabeled tape.
- How to terminate access to a labeled or unlabeled tape file.
- How to deal with errors returned by magnetic tape read and write operations.
- A complete sample program showing the use of many of the features described in this section for writing or accessing data on an unlabeled tape.

You use DEFINEs to specify attributes for labeled tapes, although DEFINEs can also be used for unlabeled tapes. This section contains some specific examples. For a general discussion of the programmatic interface to DEFINEs, refer to Section 7, Using DEFINEs. For a discussion of DEFINEs at the TACL level, refer to the *Guardian User's Guide*.

Accessing Magnetic Tape: An Introduction

Programmatic access to magnetic tapes is provided by the file-system procedures listed below:

AWAITIO[X]	Waits for the completion of outstanding I/O operations pending on the open magnetic tape unit when operating in nowait mode.
CONTROL	Controls tape positioning and rewind operations. Forward and backward positioning by record or file are supported. See Table 12-1.
FILE_CLOSE_	Terminates access to an open magnetic tape unit.
FILE_GETINFO_	Provides error information and characteristics about an open magnetic tape unit.
FILE_OPEN_	Establishes communication with the magnetic tape unit.
READ[X]	Reads records from magnetic tape.
SETMODE	Sets and clears buffered mode and streaming modes of operation, and selects tape density. See Table 12-2.
SETMODENOWAIT	Acts the same as SETMODE except that the magnetic tape functions are applied in a nowait manner.
WRITE[X]	Writes records to an open magnetic tape file.

Table 12-1 summarizes all CONTROL operations that affect magnetic tape operation.

Table 12-1. Magnetic Tape CONTROL Operations (page 1 of 2)

CONTROL Number	Operation
2	Writes an end-of-file mark.
3	Rewinds and unloads the tape without waiting for the operation to finish.
4	Rewinds the tape and takes it offline without waiting for the operation to finish (not supported on the 5130, 5160, 5170, and 5180 tape drives).
5	Rewinds the tape and leaves it online without waiting for the operation to finish.
6	Rewinds the tape and leaves it online and waits for the operation to finish.
7	Spaces forward by a given number of files.
8	Spaces backward by a given number of files.
9	Spaces forward by a given number of records.

Table 12-1. Magnetic Tape CONTROL Operations (page 2 of 2)

CONTROL Number	Operation
10	Spaces backward by a given number of records.
24	Forces an end of volume. Unloads the current volume and requests the next volume. This operation applies only to labelled tape.
26	Causes the tape process to flush all buffered records to tape

On return from one of the calls listed in Table 12-1, the condition code should be CCE if the CONTROL operation was successful. A condition code of CCL indicates an error. A condition code of CCG indicates that an end-of-file mark was encountered.

Table 12-2 summarizes all SETMODE functions that relate to magnetic tape.

Table 12-2. Magnetic Tape SETMODE Functions

SETMODE Number	Function
52	Sets and clears short write mode to allow (or disallow) write operations of less than 24 bytes.
66	Sets the tape density. This SETMODE function also has the capability of setting start/stop mode and streaming mode; however, SETMODE 119 should always be used instead. SETMODE 66 is not supported on G-series releases.
99	Sets and resets buffered mode.
119	Sets start/stop or streaming mode. SETMODE 119 is not supported on G-series releases.
120	Changes the way the tape process reports the end of file.
162	Sets compression mode for tape drives that support compression.

On return from one of the calls listed in Table 12-2, the condition code should be CCE if the SETMODE function was performed successfully. A condition code of CCL indicates an error. A condition code of CCG indicates that the attempted SETMODE function is invalid for the type of device.

For complete details of these procedure calls, CONTROL operations, and SETMODE functions, refer to the *Guardian Procedure Calls Reference Manual*.

Generally, the way you access a magnetic tape from an application program is as follows:

1. Open the file associated with the magnetic tape device (using the FILE_OPEN_ procedure). An outline is given following Step 4.
2. Position the tape to the file/record that you intend to access (using the CONTROL procedure). Refer to the next subsection, Positioning the Tape, for details.

3. Perform read or write operations on the magnetic tape (using the READ[X] or WRITE[X] procedures). See Reading and Writing Tape Records later in this section for details.
4. Terminate access to the magnetic tape (using the FILE_CLOSE_ procedure). See Terminating Tape Access later in this section for details.

The correct method for opening the magnetic tape device depends on whether the tape you access is labeled or unlabeled. A labeled tape device is always opened by passing the name of a DEFINE as the file-name parameter to the FILE_OPEN_ procedure; for example:

```
FILE^NAME ':=' "=TAPE^FILE";
LENGTH := 10;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...
```

=TAPE^FILE is assumed to be a DEFINE that describes a file on the labeled tape. Subsequent access to the file on tape is made through the file number returned in the TAPE^NUM variable.

Details about setting up the DEFINE are given with specific examples in Working With Standard Labeled Tapes later in this section.

You gain access to an unlabeled tape by opening the device by name (or logical device number) or by using a DEFINE that describes the device. The following example opens the tape device $TAPE1 by name:

```
FILE^NAME ':=' "$TAPE1";
LENGTH := 6;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...
```

Refer to Working With Unlabeled Tapes later in this section for further details about opening unlabeled tapes, including an example of using a DEFINE.

Positioning the Tape

The CONTROL procedure has several operations that move the tape backward and forward by a specific number of files or record blocks:

- Space forward a number of files using CONTROL operation 7.
- Space backward a number of files using CONTROL operation 8.
- Space forward a number of record blocks using CONTROL operation 9.
- Space backward a number of record blocks using CONTROL operation 10.
- Rewind the tape using CONTROL operation 3, 5, or 6.

Procedure calls that position by record block are valid for labeled and unlabeled tapes. The file-movement operations are redundant when dealing with labeled tapes because positioning is done using the FILESEQ and FILESECT values. The file-movement operations are necessary for accessing the desired file on unlabeled tapes.

Tape positioning by record block applies to labeled and unlabeled tapes.

Note. The tape device must be open before you can use any of the tape-positioning procedures. How you open the tape depends on whether you are using labeled or unlabeled tape. For details on opening labeled tapes, see Working With Standard Labeled Tapes later in this section. For details on opening unlabeled tapes, refer to Working With Unlabeled Tapes.

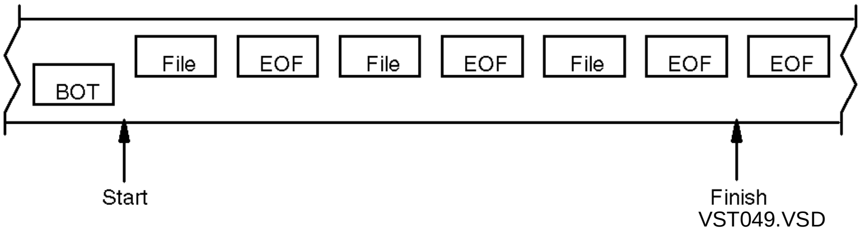
Spacing Forward and Backward by Files

CONTROL operation 7 moves the tape forward (toward the end-of-tape or EOT sticker) a specified number of files. The operation stops when the specified number of end-of-file (EOF) marks are encountered.

Note. If the number of files on the tape is less than the number specified in the call to CONTROL operation 7, then the tape will be pulled off the end of the reel. On 3480 devices, such an operation also causes an end-of-tape error (error 150).

The following example shows how to use CONTROL operation 7 to space forward by three files. The illustration assumes the following call is issued when the tape is positioned at the beginning-of-tape (BOT) sticker. Note that the tape stops immediately **after** the third EOF mark.

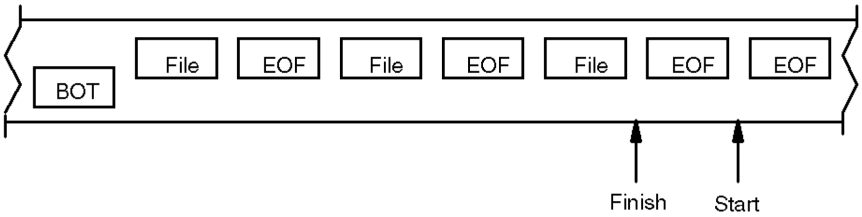
```
LITERAL SPACE^FWD^FILES = 7;
  .
  .
NUMBER^OF^FILES := 3;
CALL CONTROL(TAPE^NUM,
             SPACE^FWD^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```



CONTROL operation 8 moves the tape backward (toward the BOT sticker) a specified number of files. The operation stops either when the specified number of EOF marks are encountered or on reaching the BOT sticker.

The next example shows CONTROL operation 8 used to space the tape backward by one file, starting from the finish point of the previous example. Note that the tape stops immediately **before** the EOF mark.

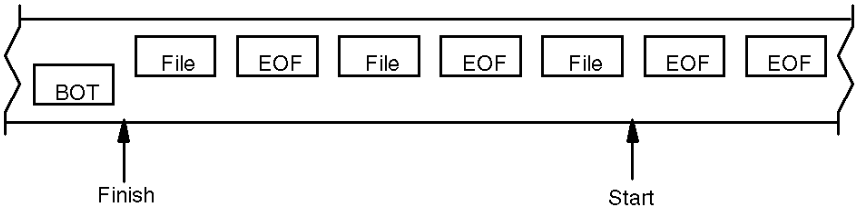
```
LITERAL SPACE^BACK^FILES = 8;
 .
 .
NUMBER^OF^FILES := 1;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```


BOT
File
EOF
File
EOF
File
EOF
EOF
Finish
Start

VST050.VSD

Then space the tape backward 10 files:

```
NUMBER^OF^FILES := 10;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^FILES,
             NUMBER^OF^FILES);
IF <> THEN
BEGIN
   CALL GET_FILEINFO_(TAPE^NUM,
                      ERROR);
   IF ERROR = 154 THEN ...                 !BOT
   ELSE ...                                !Other error
```


BOT
File
EOF
File
EOF
File
EOF
EOF
Finish
Start

VST051.VSD

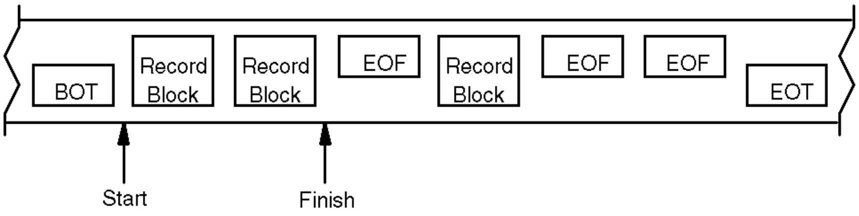
Here, because there are fewer than 10 files preceding the start position, the tape rewinds as far as BOT and the CONTROL operation returns error 154 (BOT detected when backspacing).

Spacing Forward and Backward by Record Blocks

CONTROL operation 9 moves the tape forward (toward the EOT sticker) a specified number of record blocks. This operation stops when the end of the specified number of record blocks is encountered. If an EOF mark is encountered, error 1 (EOF detected) is returned to the program and the tape stops immediately after the EOF mark. (Note that reaching the EOT sticker does not cause an error because the EOT sticker is just a warning that the physical end of the tape is near.)

The following examples show how CONTROL operation 9 moves the tape forward. The first example starts at the BOT sticker and moves the tape forward two record blocks:

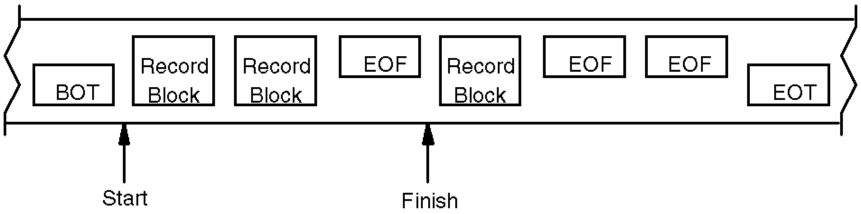
```
LITERAL SPACE^FWD^RECORDS = 9;
  .
  .
NUMBER^OF^RECORDS := 2;
CALL CONTROL(TAPE^NUM,
             SPACE^FORWARD^RECORDS,
             NUMBER^OF^RECORDS);
IF <> THEN ...
```



VST052.VSD

The next example also starts at the BOT sticker and then tries to space forward 10 record blocks. The operation stops because there are fewer than 10 record blocks in the file. The CONTROL procedure returns error 1 and the tape stops immediately after the EOF mark:

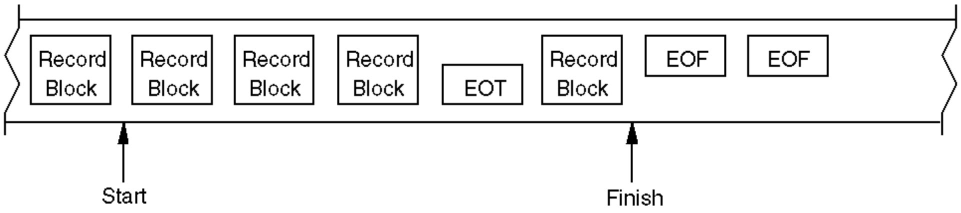
```
NUMBER^OF^RECORDS := 10;
CALL CONTROL(TAPE^NUM,
             SPACE^FWD^RECORDS,
                NUMBER^OF^RECORDS);
IF <> THEN ...
```


BOT
Record Block
Record Block
EOF
Record Block
EOF
EOF
EOT
Start
Finish

VST053.VSD

The next example shows the magnetic tape spacing forward a number of record blocks beyond the EOT sticker. The forward spacing continues because the tape has not yet reached the end of the file:

```
NUMBER^OF^RECORDS := 4;
CALL CONTROL(TAPE^NUM,
             SPACE^FWD^RECORDS,
             NUMBER^OF^RECORDS);
IF <> THEN ...
```


Record Block
Record Block
Record Block
Record Block
EOT
Record Block
EOF
EOF
Start
Finish

VST054.VSD

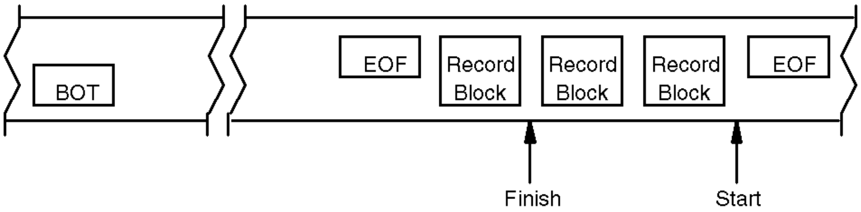
CONTROL operation 10 moves the tape backward (toward the BOT sticker) a specified number of record blocks. This operation stops when either the tape has spaced backward over the specified number of record blocks or an EOF mark or BOT sticker is encountered. Encountering an EOF mark means that the tape has rewound to the beginning of the file; the tape is positioned immediately **before** the EOF mark. Encountering the BOT sticker means that the tape has rewound to the beginning of the tape; the tape stops immediately **after** the BOT sticker, and error 154 is returned to the program.

Note. To achieve better performance, you should avoid backspacing by record blocks on the 3209/5120 tape subsystems.

The following examples show how to space backward by record blocks using CONTROL operation 10. The first example spaces the tape backward two record blocks from an initial tape position just before EOF:

```
LITERAL SPACE^BACK^RECORDS = 10;
  .
```

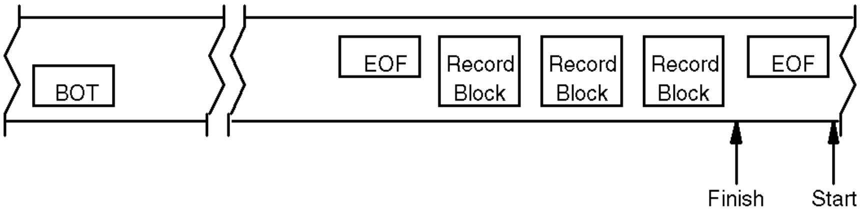
```
 .
NUMBER^OF^RECORDS := 2;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^RECORDS,
             NUMBER^OF^RECORDS);
IF <> THEN ...
```


BOT
EOF
Record Block
Record Block
Record Block
EOF
Finish
Start

VST055.VSD

If the tape is positioned immediately after the EOF mark (for example, following a space file forward operation), CONTROL operation 10 causes the tape to move to just **before** the same EOF mark. Error 1 is returned:

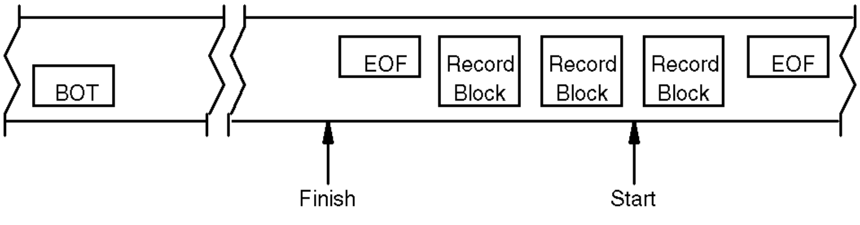
```
NUMBER^OF^RECORDS := 5;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^RECORDS,
             NUMBER^OF^RECORDS);
IF <> THEN ...
```


BOT
EOF
Record Block
Record Block
Record Block
EOF
Finish
Start

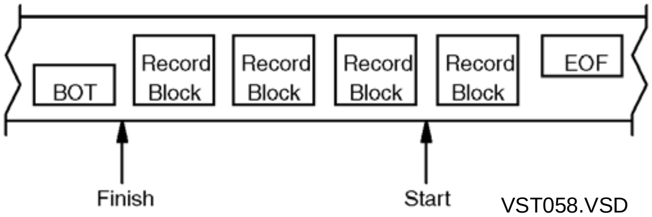
VST056.VSD

Similarly, the next example tries to space backward eight record blocks when the tape is positioned somewhere in the middle of the file and there are fewer than eight record blocks between the current tape position and the beginning of the file. The tape stops immediately **before** the EOF mark, and error 1 is returned to the program:

```
NUMBER^OF^RECORDS := 8;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^RECORDS,
                 NUMBER^OF^RECORDS);
IF <> THEN ...
```


BOT
EOF
Record Block
Record Block
Record Block
EOF
Finish
Start
VST057.VSD

The next example tries to space backward five record blocks but encounters the BOT sticker. The tape stops immediately **after** the BOT sticker, and error 154 is returned to the program:


BOT
Record Block
Record Block
Record Block
Record Block
EOF
Finish
Start
VST058.VSD

```
N
C

I
```

Rewinding the Tape

The CONTROL procedure supports several operations that enable your program to rewind a tape. You can choose to have the tape stop at the BOT sticker or unload completely. You can choose to wait for the operation to finish or continue without waiting for completion.

Use CONTROL operation 3 to rewind and unload the tape without waiting for completion:

```
LITERAL REWIND^AND^UNLOAD = 3;
  .
  .
CALL CONTROL(TAPE^NUM,
                 REWIND^AND^UNLOAD);
IF <> THEN ...
```

Use CONTROL operation 4 to rewind the tape and take the tape offline without waiting for completion. The tape stops at the BOT sticker.

```
LITERAL REWIND^OFFLINE = 4;
  .
  .
CALL CONTROL(TAPE^NUM,
                 REWIND^OFFLINE);
IF <> THEN ...
```

Use CONTROL operation 5 to rewind the tape to the BOT sticker, leaving the tape online. This procedure call does not wait for completion.

```
LITERAL REWIND^ONLINE = 5;
 .
 .
CALL CONTROL(TAPE^NUM,
             REWIND^ONLINE);
IF <> THEN ...
```

Finally, CONTROL operation 6 rewinds the tape to BOT and leaves the tape online. Your program waits for the operation to finish.

```
LITERAL REWIND^AND^WAIT = 6;
 .
 .
CALL CONTROL(TAPE^NUM,
             REWIND^AND^WAIT);
IF <> THEN ...
```

Reading and Writing Tape Records

Application programs read and write tape records by calling the READ[X] and WRITE[X] procedures. A record is the amount of data that is read by a single read operation or written by a single write operation. A record block can be as large as 57,344 bytes depending on the particular tape device. The shortest record block is usually 24 bytes, but that can be changed using SETMODE function 52 for some controllers, as described later.

Before performing either read or write operations, the tape file must already be open. The way you do this depends on whether you are accessing a labeled or unlabeled tape. Refer to Working With Standard Labeled Tapes later in this section for information on how to open a labeled tape file and to Working With Unlabeled Tapes, also later in this section, for details on how to open a tape file for unlabeled tape access.

The following paragraphs describe how to read and write tape records.

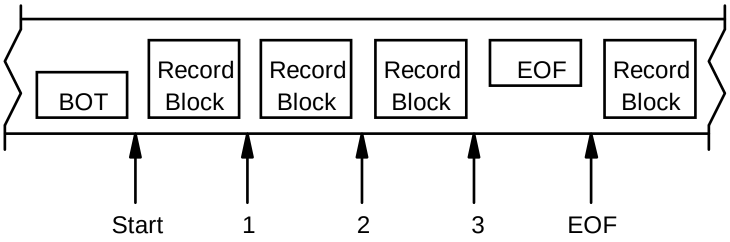
Reading Tape Records

Use the READ[X] procedure to read record blocks from magnetic tape. One READ[X] procedure call reads one record block from the tape. Whenever a read operation is issued against the tape file, the tape spaces forward one record block, even if the read is for zero bytes.

An example shows how to read from a magnetic tape. Consider a file on tape that consists of three record blocks, where each record block contains 1024 bytes. Repeated reads of 2048 bytes are executed as follows:

```
LITERAL EOF = 1;
INT LOOP := 1;
 .
 .
WHILE LOOP = 1 DO
BEGIN
   RCOUNT := 2048;
   CALL READX(TAPE^NUM,SBUFFER,
              RCOUNT,COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPE^NUM,
                         ERROR);
      IF ERROR = EOF THEN LOOP := 0
      ELSE ........;
   END
   ELSE
   BEGIN

      !Process the record block returned in BUFFER.

   END;
END;
```

The first, second, and third reads each transfer 1024 bytes into SBUFFER, return 1024 in COUNT^READ, and set the condition code to CCE (the no-error condition code). The fourth read operation encounters an EOF mark; nothing is transferred into SBUFFER, 0 is returned in COUNT^READ, and the condition code is set to CCG. The FILE_GETINFO_ procedure returns 1 in the error variable, informing the process that the EOF mark is reached.



CDT 059.CDD

If the value passed in the *read-count* parameter is not enough to read an entire record block, an error indication is returned to the application. For example, a record block on tape contains 1024 bytes of data and a read of 256 bytes is requested:

```
RCOUNT:= 256;
CALL READX (TAPE^NUM,
            SBUFFER,
            RCOUNT,
            COUNT^READ);
IF < THEN                        !Error encountered
BEGIN
   CALL FILE_GETINFO_(TAPE^NUM,
                      BUFFER);
    .
    .
```

256 bytes are transferred into SBUFFER, 256 is returned in COUNT^READ, and the condition code is set to CCL. The call to FILE_GETINFO_ returns error number 21 (illegal count specified). After the read operation, the tape is positioned immediately before the beginning of the next record block on tape.

Writing Tape Records

Use the WRITE[X] procedure to write record blocks to magnetic tape. Each WRITE[X] procedure call writes one record block to the tape. The WRITE[X] procedure is typically used when sequentially appending information on the tape.

The following procedure call writes one record block to tape:

```
WCOUNT := 2048;
CALL WRITEX(TAPE^NUM,
            TAPE^BUF,
            WCOUNT,
            COUNT^WRITTEN);
IF <> THEN ...
```

Here, 2048 bytes are written to tape. The value 2048 is returned in the COUNT^WRITTEN variable.

Normally, the file system pads write operations of fewer than 24 bytes with null (0) characters. The number of bytes of null characters is 24 minus the write count specified in the WRITE[X] procedure call. Therefore the smallest record block that can be written to a tape is 24 bytes.

Using SETMODE function 52, an application can either disallow writing record blocks that are shorter than 24 bytes, allow records that are shorter than 24 bytes but pad them with null characters, or allow records that are shorter than 24 bytes without padding. If writing record blocks shorter than 24 bytes without padding is allowed, then the limit on the shortest record size allowed is controller-dependent.

The following example disallows writing record blocks that are shorter than 24 bytes:

```
LITERAL SHORT^WRITE^MODE = 52,
        NO^SHORT^WRITES  = 1;
  .
  .
CALL SETMODE(TAPE^NUM,
             SHORT^WRITE^MODE,
             NO^SHORT^WRITES);
IF <> THEN ...
```

If an application disallows writing short record blocks but later tries to write a record block of fewer than 24 bytes, the WRITE[X] procedure returns error 21.

When the application has finished writing record blocks to an unlabeled tape, the application should indicate the end of the tape file by writing an EOF mark to the tape. You do this by calling CONTROL operation 2:

```
LITERAL WRITE^EOF = 2;
  .
```

```
 .
CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...
```

Note that, for unlabeled tapes, closing a file does not write an EOF mark.

For labeled tapes, the end of the tape is identified by the tape labeling mechanism. All the application needs to do is to detect the EOT sticker, stop writing, and close the file.

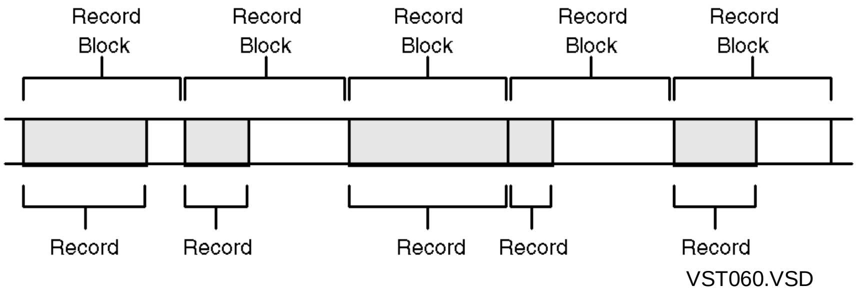
Blocking Tape Records

A record is a collection of related information as seen by the application; for example, a data structure containing an account number, name, and balance. Records can be fixed or variable length.

A record block contains the data that is written to or read from tape in one read or write operation. Record blocks within a tape file are typically the same length. A record block, however, can be larger than a record as recognized by the application.

The relationship between the record and the record block written to tape depends on whether the record length is fixed or unspecified. If you are using an unspecified record length, each record block typically contains one record and is padded with blank space. The size of the record block determines the maximum size of the record. Figure 12-1 shows this concept.

Figure 12-1. Physical Tape Records Containing Records of Unspecified Length



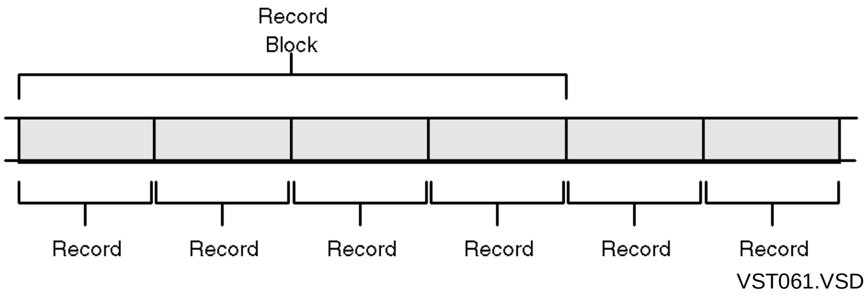
If your application uses fixed-length records, then for efficiency, you can block multiple records into one record block that is read from or written to tape in one operation. The record-block size must be an exact multiple of the record size. Figure 12-2 shows this concept.

Blocking records is more efficient than having equal-sized physical and records for the following reasons:

- Each physical read or write operation has an overhead associated with it. Blocking reduces the number of physical read and write operations, therefore reducing the overhead.
- Record blocks are separated from each other on tape by an interrecord gap. Blocking reduces the number of interrecord gaps that are needed and therefore uses less magnetic tape to store the same information.

When the application performs a read operation, however, it receives a record block of data and must therefore deblock the record block to extract the record that it wants to read.

Figure 12-2. Physical Tape Records Containing Records of Fixed Length



Working in Buffered Mode

We recommend using buffered mode to improve the performance of tape read and write operations.

In buffered mode, the tape process, which is a system process, replies to write requests as soon as the data has been transferred from the application to the tape process buffer. The application can then continue while the data from the previous request is written to tape. When buffering, the request that returns the error is not executed.

If the application continues to issue write requests only, the tape process accepts the requests until the buffer is full. When the buffer is full, the tape process holds further requests until previous requests finish.

The application cannot determine which of its previous write requests have been written to tape. If an error occurs in one of the previous write requests, an error condition is returned to the application but the application cannot determine which request failed.

Usually, the tape process does not process requests other than write requests until all previous write requests have been completed. For example, if the application program issues a CONTROL request against the tape file, the tape process holds the request and the application waits until all outstanding writes have finished successfully. Successful completion of a non-write request means that all records have been written to tape without error.

Some tape devices, such as the 5170, do support buffered end-of-file marks. This feature must be specifically enabled using SETMODE function 99 as described below.

If the application closes the tape device before the previous write requests have finished, the tape process performs the outstanding requests before the close request is finished. However, because the FILE_CLOSE_ procedure does not return regular file-system errors, an application closing the tape process when buffers are awaiting completion cannot be sure those data records have been successfully written to tape.

Note. recommends setting a tape device to unbuffered mode before closing. Doing so ensures that all data records have been successfully written to tape.

If the application issues a close request against an unlabeled tape while write requests are still outstanding, then the data on tape is not terminated with an end-of-file mark. Any application that later tries to read the tape will not encounter the conventional two end-of-file marks that indicate the logical end of the tape. The application might therefore encounter a runaway tape condition or incorrect data.

The above situation can occur when the system closes the tape file for an application that is stopping or abending. Also, note that this is no different from unbuffered operation. The tape process does not write trailing end-of-file marks for an application that accesses unlabeled tape.

When an application encounters an end-of-tape error, the application must stop writing records and send the end-of-tape sequence. The EOT error is returned for write and write-end-of-file-mark requests only.

An error indication can be reporting either a problem with the request itself or a problem with some previous buffered write. If an error is reported on a previous operation, the request that resulted in the error is not performed. Requests that terminate a write sequence (typically CONTROL requests such as write-end-of-file-mark) must allow for this.

There are no special application considerations for read sequences. Buffered-mode reads are handled by the tape process. If buffered mode is enabled, the tape process responds to an application read request by reading ahead of the requested record. The requested record (or error condition) is returned to the application; the remainder of the data read from tape remains in the tape buffer.

All other requests are performed in buffered mode as they are in unbuffered mode, except that they can return errors from previous I/O operations.

Invoking and Revoking Buffered-Mode Operation

Buffered mode is allowed only on an exclusive basis: only a single opener can have a device open at any one time if buffered mode is to be enabled.

Buffered mode is disabled by default.

Buffered mode is enabled by an exclusive opener through SETMODE operation 99; for example:

```
LITERAL   BUFFERED^MODE = 99,
          ENABLED       =  1,
          DISABLED      =  0;
  .
  .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ENABLED);
```

The FILE_OPEN_ request need not specify exclusive access in the call; the tape process enforces exclusive access by disallowing further opens if buffered mode is enabled and by disallowing buffered mode if there is more than a single opener. Error 12 (file in use) is returned if the SETMODE request is rejected for this reason.

The SETMODE call to enable buffered-mode operation also fails if the tape process cannot allocate an I/O segment for its buffers. If the allocation fails, operation continues in unbuffered mode and error 33 (no buffer space) is returned by the SETMODE call.

Buffered mode remains enabled until the application closes the device or disables buffered mode by calling SETMODE; for example:

```
LITERAL   BUFFERED^MODE = 99,
          ENABLED       =  1,
          DISABLED      =  0;
  .
  .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             DISABLED);
```

If buffered mode is disabled using SETMODE while buffered writes are waiting to be written to tape, the SETMODE does not finish until all outstanding write operations have completed or an error occurs. Thus, a SETMODE during a buffered write sequence behaves like any request other than a write request: an error is returned to indicate that a previous write operation did not finish successfully.

The SETMODE request to enable buffered mode, when issued while already operating in buffered mode, can serve as a checkpoint operation to allow an application to confirm that all previous write operations have finished successfully and that the data was written to tape. This is true of any request (except WRITE[X]) issued after a sequence of WRITE[X] requests. There is no explicit request that is defined for this checkpointing purpose.

In all cases, the application must check the error code returned from the SETMODE request to be sure of its success.

Flushing the Buffer

You can cause the tape process to write the contents of its buffer to tape and synchronize the drive by issuing the CONTROL 26 operation. You can use this operation following a write or a write end-of-file mark operation:

```
LITERAL SYNCHDATA = 26;
  .
  .
CALL CONTROL(TAPE^NUM,
             SYNCHDATA);
```

The operation finishes when the synchronization is complete.

Buffered Mode for Streaming Devices (D-Series Only)

With streaming devices such as the 5120 cartridge tape, the device itself (or its integral formatter) buffers requests in a way similar to buffered mode. The device replies to the controller with a positive acknowledgment as soon as the data is transferred to the device buffer, not when the data has been written to tape. Streaming implies file-level, rather than record-level, recovery.

For devices such as the 5120 cartridge, buffered mode is enabled by default whenever a device is placed into streaming mode by the following call:

```
LITERAL STREAMING^MODE = 119,
        ENABLED        =   1,
        DISABLED       =   0;
  .
  .
CALL SETMODE(TAPE^NUM,
             STREAMING^MODE,
             ENABLED);
```

For other streaming devices, such as the 5170, 5180 and 5190 drives, buffering must be explicitly enabled using SETMODE function 99 as described for non-streaming devices. For these devices, buffering is not automatic when streaming mode is enabled using SETMODE function 119.

Buffering End-of-File Marks

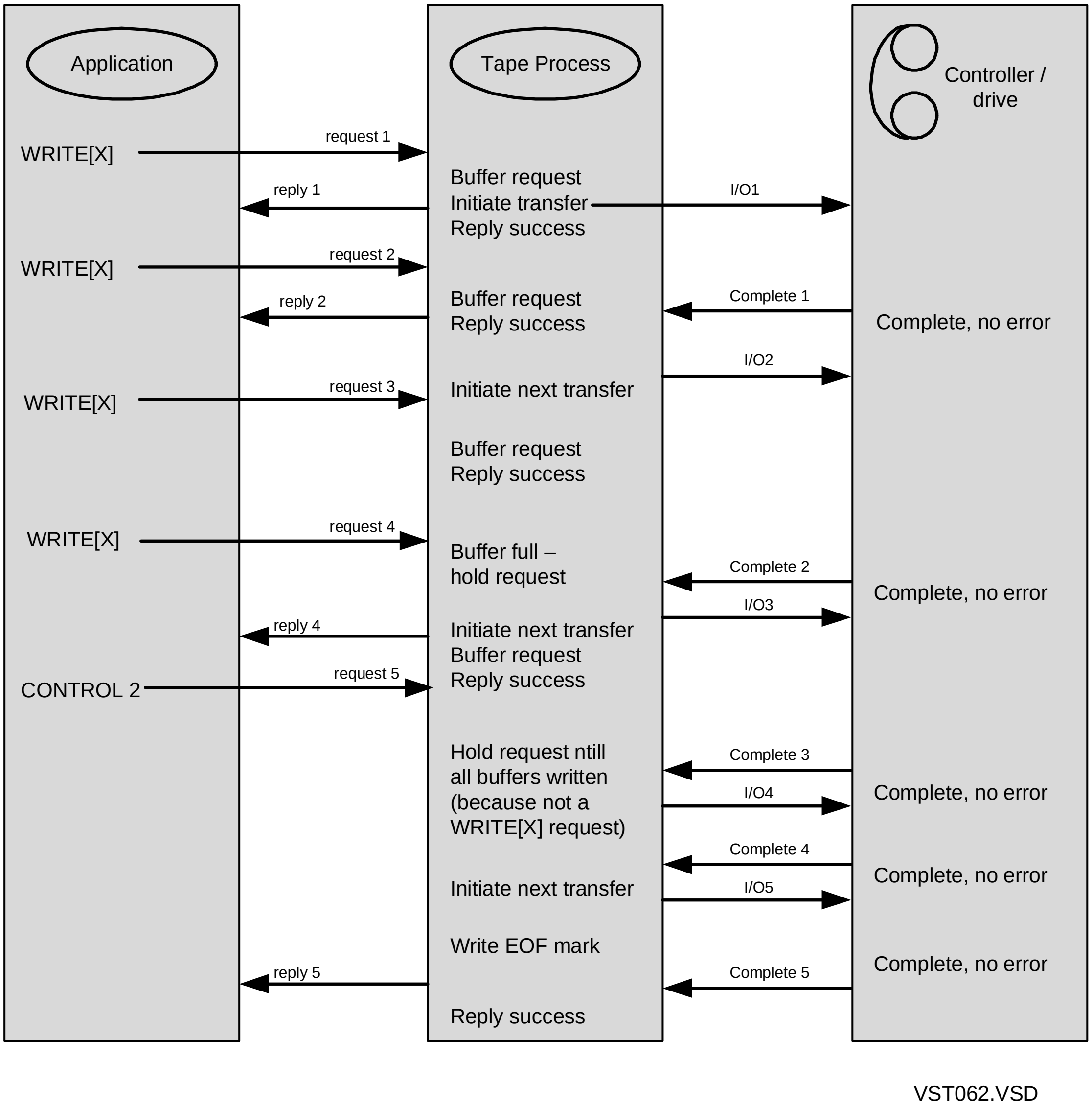
Some tape devices, such as the 5170, allow end-of-file marks to be buffered. To achieve EOF mark buffering, you use SETMODE function 99 with *`parameter-1`* set to 2:

```
LITERAL   BUFFERED^MODE            = 99,
          ENABLE^BUFFERED^EOFMARKS =  2;
  .
  .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ENABLE^BUFFERED^EOFMARKS);
```

An Example of Buffered-Mode Operation

Figure 12-3 shows an example of a buffered-mode write sequence. In this example, the application sends four write requests and an EOF mark to the tape process. In this case, three write requests are enough to fill the tape buffer. The application must therefore wait for the fourth write request (Step 13) until one of the buffered write operations is written to tape. Once the request to write an EOF mark is issued (Step 19), the tape process holds this request until all outstanding transfers to tape are complete.

Figure 12-3. Example of Buffered-Mode Operation


Application
Tape Process
Controller / drive
WRITE[X]
request 1
Buffer request
Initiate transfer
Reply success
reply 1
I/O1
WRITE[X]
request 2
Buffer request
Reply success
reply 2
Complete 1
Complete, no error
I/O2
WRITE[X]
request 3
Initiate next transfer
Buffer request
Reply success
WRITE[X]
request 4
Buffer full –
hold request
Complete 2
Complete, no error
I/O3
reply 4
Initiate next transfer
Buffer request
Reply success
CONTROL 2
request 5
Hold request ntill
all buffers written
(because not a
WRITE[X] request)
Complete 3
I/O4
Complete, no error
Complete 4
Complete, no error
Initiate next transfer
I/O5
Write EOF mark
Complete, no error
reply 5
Complete 5
Reply success
VST062.VSD

Working With Standard Labeled Tapes

The operating system provides support for magnetic tapes written with standard ANSI or IBM labels. The labeling mechanism allows for easy transfer of information between systems from different vendors using magnetic tape.

Both the ANSI and IBM standards use labels to describe tape volumes, files, and file sections. Here, a tape volume is a complete tape reel; a file is a file of information written to the tape; for large files, a file section identifies the part of a file that resides on a given tape volume. The concept of a file section therefore makes it possible to have tape files that occupy more than one tape reel.

For full details of the ANSII standard see the ANSI X31.27-1987 standard as described in “File Structure and Labeling of Magnetic Tapes for Information Interchange” published by the American National Standards Institute. For the layout of the IBM and ANSI label structures, see the *Guide to Common System Operation Tasks*.

Enabling Labeled Tape Processing

To enable labeled tape processing, your system must be set up as follows:

- Tape-label processing must be turned on at system-generation time. Refer to the *System Generation Manual for Disks and Tapes* for details.
- The labeled tape server process $ZSVR must be running. This process is typically started during cold load.

HP sites can create their own labeled tapes, or they can work with tapes written at any site with any equipment so long as the tape contains standard ANSI or IBM labels.

Creating Labeled Tapes

Labeled tapes created at a HP site must first be initialized by the MEDIACOM utility or the TAPECOM utility, which put volume labels on the tape and an indication that this is a scratch tape. Refer to the *DSM/TC Operator Interface (MEDIACOM)* for details on the MEDIACOM utility; refer to the *Guardian Disk and Tape Utilities Reference Manual* for details on the TAPECOM utility.

Checking for Labeled Tape Support

You can check whether labeled tape support is turned on by calling the LABELEDTAPESUPPORT procedure:

```
RETURNED^VALUE := LABELEDTAPESUPPORT;
IF RETURNED^VALUE = 0 THEN CALL PROCESS_STOP_
ELSE ....

     !continue labeled tape processing
```

The value returned by the LABELEDTAPESUPPORT procedure is 0 if tape-label processing is **not** turned on or 1 if tape-label processing **is** turned on.

Accessing Labeled Tapes

You gain access to a labeled tape by passing a tape DEFINE name to the FILE_OPEN_ procedure. $ZSVR responds to the FILE_OPEN_ call by sending a message to the operator to mount the tape on any tape drive. (Recall that when handling labeled tapes, it is not necessary to identify the device.)

```
ERROR := FILE_OPEN_(=TAPE^FILE^1,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...
```

When handling labeled tapes, you use a different DEFINE for each file that exists on a labeled tape. =TAPE^FILE^1 identifies one such file.

You create the DEFINE either interactively at the TACL prompt (see the *Guardian User's Guide*) or programmatically using procedure calls. Complete details on how to programmatically create DEFINEs are given in Section 7, Using DEFINEs.

You set DEFINE attributes for opening the tape file depending on several factors, such as whether you are creating a new file or reading or updating an existing file; how many tape volumes the file takes; or how many files are stored on the one tape volume.

Recall from Section 7, Using DEFINEs:

- Before working with DEFINEs, you must enable DEFINEs by issuing a DEFINEMODE procedure call:

  ```
  NEW^VALUE := 1;
  CALL DEFINEMODE(NEW^VALUE,
                  OLD^VALUE);
  ```

- Once you have specified the DEFINE attributes in the working set, you create the DEFINE using the DEFINEADD procedure:

  ```
  DEFINE^NAME ':=' "=NAME^OF^DEFINE         ";
  CALL DEFINEADD(DEFINE^NAME);
  ```

The following paragraphs describe the DEFINE attributes most commonly found in tape DEFINEs.

In addition, the following paragraphs include instructions for setting magnetic tape parameters that cannot be set using DEFINEs. These include setting buffered mode and choosing the device mode. These parameters are set using the SETMODE procedure **after** the tape file is open.

Specifying the DEFINE CLASS

Before setting any other DEFINE attributes for magnetic tape, you must first set the DEFINE class to “TAPE.” Doing so sets the default attributes in the working set to the default values for tape. You set the DEFINE class using the DEFINESETATTR procedure as follows:

```
ATTRIBUTE^NAME ':=' "CLASS           ";   !16 bytes
VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

Remember from Section 7, Using DEFINEs, that we recommend supplying the current default values for volume and subvolume to every call to DEFINESETATTR. Some attributes need these values; others do not. However, supplying these values never does any harm.

Specifying the Label Type

A labeled tape DEFINE must have the LABELS attribute set to ANSI, IBM, or BYPASS. (The BACKUP and IBMBACKUP attributes are reserved for use by the BACKUP and RESTORE utilities.) You can set this attribute programmatically using the DEFINESETATTR procedure. You must set this attribute to the same value as in the tape label for your program to access the tape. The $ZSVR process displays a message prompting for an appropriate tape volume. Your program will wait until an appropriate tape is mounted before proceeding.

An example follows:

```
ATTRIBUTE^NAME ':=' "LABELS          ";   !16 bytes
VALUE ':=' "ANSI" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

This call sets the label type to “ANSI,” identifying the tape as containing standard ANSI labels.

Specifying Volume and File

You do not need to specify the name of the tape device when accessing labeled tape. Instead, you identify the file you want by specifying the name of the tape volume and the name of the file on that volume. The tape can therefore be mounted on any device and the system will find it. If the tape is mounted on a tape drive on a remote system in the network, you also need to specify the system name in the SYSTEM attribute.

You specify the volume name using the VOLUME DEFINE attribute. (The VOLUME attribute is optional when the USE attribute is set to “OUT.”) An example follows:

```
ATTRIBUTE^NAME ':=' "VOLUME          ";   !16 bytes
VALUE ':=' "XT55" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

This call to DEFINESETATTR sets the volume name to “XT55.” This call identifies the tape spool containing the file you want to access. The volume name is embedded in the volume label written at the beginning of the tape.

Now use the FILEID attribute to identify the file within volume XT55. If you are reading or appending to an existing file, then the FILEID must exactly match the FILEID given to the file when the file was created. If you are creating a new file, you can use any FILEID that is unique on the tape volume.

The following example shows how to set the FILEID value:

```
ATTRIBUTE^NAME ':=' "FILEID          ";   !16 bytes
VALUE ':=' "FILE1" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

If you are accessing a volume that contains more than one file, you also need to specify the file sequence number. Use the FILESEQ DEFINE attribute as follows:

```
ATTRIBUTE^NAME ':=' "FILESEQ         ";   !16 bytes
VALUE ':=' "1";
LENGTH := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       LENGTH,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

Specifying the I/O Operation

You must specify the type of I/O operation you want to perform on the labeled tape. You can write a new file, read from an existing file, or append to an existing file. You can specify the I/O operation using either the *access* parameter of the FILE_OPEN_ procedure or the USE DEFINE attribute.

To specify reading, either set the *access* parameter of the FILE_OPEN_ procedure for read-only access or set the USE attribute to "IN." In either case, when the file is opened, the tape is positioned immediately before the first record in the file.

For writing a new file, you can either set the access parameter of the FILE_OPEN_ procedure for write-only access or set the USE attribute to "OUT." When the open finishes, an empty file is created ready to be written.

Appending can be specified only by setting the USE attribute to "EXTEND." When the file is opened, the tape is positioned at the end of the file ready for appending. The file must be the last file in a file set or an error condition is returned.

An example of how to set the USE attribute follows:

```
ATTRIBUTE^NAME ':=' "USE            ";   !16 bytes
VALUE ':=' "EXTEND" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

Selecting the Conversion Mode

Tapes with IBM labels are encoded in EBCDIC notation. Information taken from these tapes must be converted into ASCII code before the HP system can use it. Conversely, data written to a tape with IBM labels must be converted from ASCII code to EBCDIC code before being written to tape.

Use the EBCDIC DEFINE attribute to perform code conversion. You must specify this attribute whenever you access a tape with IBM labels (the LABELS attribute is set to "IBM"). Set the EBCDIC attribute to "IN" for reading from the tape file or "OUT" for writing or appending to the tape file; for example:

```
ATTRIBUTE^NAME ':=' "EBCDIC         ";   !16 bytes
VALUE ':=' "IN" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

ANSI tapes are already in ASCII form and therefore do not need converting.

Specifying the Block and Record Sizes

The BLOCKLEN, RECLEN, and RECFORM attributes allow you to specify how **records** are blocked into **record blocks**. Recall from Blocking Tape Records, earlier in this section, that a record is a collection of related information as seen by the application, and that one or more records can be blocked into one record block that is read from or written to tape in one operation.

You must specify the relationship between record blocks and records using the BLOCKLEN, RECLEN, and RECFORM attributes. These attributes are required to provide compatibility with other vendors. The blocking and deblocking of records is done programmatically, as described earlier in Blocking Tape Records.

When writing a file, you must specify these values to correspond to the record size and block size you will use when writing blocks to the file. When reading, the tape process always uses the values for BLOCKLEN, RECLEN, and RECFORM that are written in the tape label; the values in the DEFINE are not checked and don’t need to match the values in the tape label.

The attributes are described as follows:

- The BLOCKLEN attribute specifies the record-block size in bytes. When appending to a tape file, you must specify the same value as contained in the file label. If you are creating a new tape file, you must set the value to either an exact multiple of the record length if you are using fixed-length records or a value equal to the maximum record length if you are using unspecified-length records. The following example sets the BLOCKLEN attribute:

  ```
  ATTRIBUTE^NAME ':=' "BLOCKLEN        ";   !16 bytes
  VALUE ':=' "2048" -> @S^PTR;
  ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                         VALUE,
                         @S^PTR '-' @VALUE,
                         DEFAULT^NAMES);
  IF ERROR > 0 THEN ...
  ```

- The RECLEN attribute specifies the length of the record if you are using fixed-length records. For appending, you must set this value to the corresponding value in the file label. This attribute must not be specified if you are using variable-length records.

  ```
  ATTRIBUTE^NAME ':=' "RECLEN          ";   !16 bytes
  VALUE ':=' "64" -> @S^PTR;
  ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                         VALUE,
                         @S^PTR '-' @VALUE,
                         DEFAULT^NAMES);
  IF ERROR > 0 THEN ...
  ```

2 The RECFORM attribute specifies whether records are fixed length or variable length. It has a value “F” for fixed or “U” for undefined (variable). The following example sets the RECFORM attribute:

```
ATTRIBUTE^NAME ':=' "RECFORM         ";   !16 bytes
VALUE ':=' "F";
LENGTH := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       LENGTH,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

Specifying Tape Density

When writing data, to tape you can specify the tape density in bits per inch. The valid densities are 1600 and 6250. By default, the system uses the density configured for the tape drive at system-generation time. You do not need to set the tape density on reading the tape; the tape controller automatically uses the density of the tape.

Specifically, the criteria for establishing the tape density are as follows:

- If the user specifies the density, then the tape process uses that density.
- Otherwise, if the tape is labeled, then the tape process uses the density indicated in the tape label.
- Otherwise, the tape process uses the tape density assigned to the device during system generation.

Use the DENSITY attribute to set the tape density. The following example sets the tape density to 1600 bits per inch:

```
ATTRIBUTE^NAME ':=' "DENSITY         ";   !16 bytes
VALUE ':=' "1600" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       VALUE,
                       @S^PTR '-' @VALUE,
                       DEFAULT^NAMES);
IF ERROR > 0 THEN ...
```

Specifying Other Tape DEFINE Attributes

In addition to the CLASS TAPE DEFINE attributes already discussed, you can specify the following:

- The tape device. Use the DEVICE attribute. Note that it is usually not necessary to specify the tape device when accessing a labeled tape. This attribute is commonly used with unlabeled tape processing.
- The earliest date at which the tape can be overwritten. Use the EXPIRATION attribute to specify the actual date or RETENTION to specify a number of days.

You will get an error if you try to write to the file before the expiration date. However, you can still read the file after the expiration date.

- A generation group for the file and a version number within the generation group. Use the GEN attribute to identify the generation group and the VERSION attribute to specify the version.

  The generation group identifies a specific instance of a file. For example, a payroll application might create a new instance of the payroll file each time the file is updated. The GEN attribute allows you to identify a specific generation.

  The version number allows you to identify a specific instance of a generation. For example, suppose the payroll application stopped part way through writing out the payroll file and had to start over. You should not create another generation because the data belongs to the same generation as the file that was not complete. Instead, you create a different version of the same generation.

  The HP tape process does not check the generation group and version numbers when reading a file. However, you might need to specify these values when writing a file if the file is to be read on another vendor's equipment and the reading process expects the information.

- An additional message to display to the operator when the DEFINE is opened. Use the MOUNTMSG attribute.
- The identity of the owner of the files on the volume. Use the OWNER attribute.

Selecting Device Mode (D-Series Only)

On D-series releases only, if you are accessing a tape on a drive controlled by a 3209/5120 controller, you can access the cartridge in either start/stop or streaming mode. If you are using this type of drive, you should therefore use SETMODE function 119 to specify the access mode. You set the device mode after opening the tape device but before accessing the tape.

The following example sets streaming mode.

```
LITERAL DEVICE^MODE = 119,
        START^STOP  =   0,
        STREAMING   =   1;
  .
  .
CALL SETMODE(TAPE^NUM,
             DEVICE^MODE,
             STREAMING);
IF <> THEN ...
```

A program can be designed to run for any type of tape drive/controller by using the FILE_GETINFO_ procedure immediately after opening the tape file to determine the subdevice type of the file. If the subdevice type is 5 (that is, the 3209 controller for a 5120 device), then you prompt the user for the mode and set the mode accordingly. If the subdevice type is not 5, continue without setting the device type.

Setting Buffered Mode

We recommend using buffered mode to increase throughput. However, to use buffered mode, the application must be able to recover from errors in any buffering mode it uses. See Recovering From Errors, later in this section, for details.

In the case of a write operation, the application is allowed to continue as soon as the tape process has received the write request. Without buffered mode, the application has to wait for each write to tape to finish before continuing. When reading in buffered mode, the tape process reads ahead in anticipation of sequential reads.

You turn on buffered mode using SETMODE function 99 after the tape device is open:

```
LITERAL BUFFERED^MODE = 99,
        ON            =  1,
        OFF           =  0;
 .
 .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ON);
IF <> THEN ...
```

Buffered mode gets turned off when the application closes the tape process or when the application explicitly turns off buffered mode:

```
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             OFF);
IF <> THEN ...
```

For complete details on buffered-mode operation, refer to Working in Buffered Mode, earlier in this section.

Writing to the Only File on a Labeled Tape Volume

To write to the only file on a labeled tape volume, you need to do the following:

- Create a DEFINE for opening the file for writing
- Open the DEFINE and write to it

You create the DEFINE only once, then use it whenever you write to the file. The following paragraphs describe how to create the DEFINE and how to write to the tape using the DEFINE.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.

2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
   - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”
   - The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
   - The volume identifier (VOLUME attribute). Set this value to the value written to the tape in the volume label.
   - The file identifier (FILEID attribute). If the file already exists, the file identifier must be the same as the file identifier in the file label. If the file does not yet exist, the file identifier can be any valid file identifier.
   - The file sequence number (FILESEQ attribute). This value must be set to 1 (or not specified, as the default is 1) because it is the first file on the tape.
   - The I/O operation (USE attribute). This value must be set to “EXTEND” to append records to the file or “OUT” to write a new file.
   - The conversion mode (EBCDIC attribute). Set this value to “OUT” to convert ASCII code to EBCDIC on output. Use this option only if the tape uses IBM standard labels.
   - The record type (RECFORM attribute). This value specifies fixed-length or variable-length records.
   - The record length (RECLEN attribute). If you are appending to an existing file, this value must be equal to the value written to the file label when the tape file was created.
   - The record-block length (BLOCKLEN attribute). Set this value to either a multiple of the fixed record length or the maximum variable record length. If you are creating a new file, this value is placed in the tape label. If you are appending to a file, this value must equal the corresponding value in the file label on the tape.
   - The tape density (DENSITY attribute). This value must be the same as the density of existing data on the tape.
3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =TAPEFILE^APPEND that describes a labeled tape file using standard ANSI labels:

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to ANSI:
ATTRIBUTE^NAME ':=' "LABELS          ";
ATTRIBUTE^VALUE ':=' "ANSI" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to MYVOL:
ATTRIBUTE^NAME ':=' "VOLUME          ";
ATTRIBUTE^VALUE ':=' "MYVOL" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to TAPEFILE:
ATTRIBUTE^NAME ':=' "FILEID          ";
ATTRIBUTE^VALUE ':=' "TAPEFILE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 1:
ATTRIBUTE^NAME ':=' "FILESEQ         ";
ATTRIBUTE^VALUE ':=' "1";
ATTRIBUTE^LEN := 1;

ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...
```

```
!Set the USE attribute to EXTEND:
ATTRIBUTE^NAME ':=' "USE              ";
ATTRIBUTE^VALUE ':=' "EXTEND" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECFORM attribute to F:
ATTRIBUTE^NAME ':=' "RECFORM          ";
ATTRIBUTE^VALUE ':=' "F";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECLEN attribute to 512:
ATTRIBUTE^NAME ':=' "RECLEN           ";
ATTRIBUTE^VALUE ':=' "512" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the BLOCKLEN attribute to 2048:
ATTRIBUTE^NAME ':=' "BLOCKLEN         ";
ATTRIBUTE^VALUE ':=' "2048" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the DENSITY attribute to 1600:
ATTRIBUTE^NAME ':=' "DENSITY          ";
ATTRIBUTE^VALUE ':=' "1600" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=TAPEFILE^APPEND        ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Writing to the File

Use the DEFINE created above for appending to the file as follows:

1. Open the DEFINE using the FILE_OPEN_ procedure. If the DEFINE attributes match the attributes in the tape label, then the file is opened. If you are opening the file with write-only access or with the USE attribute set to “OUT,” then the file is created and opened. The VOLUME attribute is optional when the USE attribute is set to “OUT.”

   The returned file number relates to the tape drive that the tape is mounted on.

2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Write records to the file using the WRITEX procedure.

The following code fragment writes to the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to block four records into one record block before the record block is written to tape in one write operation.

```
LITERAL BUFFERED^MODE = 99,
        ON = 1;
 .
 .
!Open the tape file:
FILE^NAME ':=' "=TAPEFILE^APPEND" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,ON);
IF <> THEN ...
 .
 .
!Block four records into the output buffer:
SBUFFER[0]    ':=' LOGICAL^BUFFER^1[0] FOR 512;
SBUFFER[512]  ':=' LOGICAL^BUFFER^2[0] FOR 512;
SBUFFER[1024] ':=' LOGICAL^BUFFER^3[0] FOR 512;
SBUFFER[1536] ':=' LOGICAL^BUFFER^4[0] FOR 512 -> @S^PTR;

!Write a record block to the tape file:
CALL WRITEX(TAPE^NUM,SBUFFER,
            @S^PTR '-' @SBUFFER);
 .
 .
```

Writing to a File on a Multiple-File Labeled Tape Volume

If the labeled tape contains multiple files, the procedure for writing records to the file is similar to that for writing to the only file on a labeled tape volume. Again you need to create a DEFINE and then use the DEFINE for writing to the file. The difference is that the FILESEQ attribute specified in the DEFINE must identify the correct file.

The following paragraphs show how to create a DEFINE for, and write records to, a file on a multiple-file labeled tape.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.
2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
    - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”
    - The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
    - The volume identifier (VOLUME attribute). Set this value to the value written to the tape in the volume label.
    - The file identifier (FILEID attribute). If the file already exists, the file identifier must be the same as the one in the file label. If the file does not yet exist, you can specify any valid file identifier.
    - The file sequence number (FILESEQ attribute). This value must be set to the sequence number of the file on the tape. For example, if the DEFINE will describe the seventh file on the tape, then the FILESEQ attribute must be set to 7.
    - The I/O operation (USE attribute). This value must be set to “EXTEND” to append records to the file or “OUT” to write records in a new file.
    - The conversion mode (EBCDIC attribute). Set this value to “OUT” to convert ASCII code to EBCDIC on output. Use this option only if the tape uses IBM standard labels.
    - The record type (RECFORM attribute). This value specifies fixed-length or variable-length records.
    - The record length (RECLEN attribute). If you are appending to an existing file, this value must be equal to the value written to the file label when the tape file was created.
    - The record-block length (BLOCKLEN attribute). Set this value to either a multiple of the fixed record length or the maximum variable record length. If you are creating a new file, this value is placed in the tape label. If you are appending to a file, this value must also equal the corresponding value in the file label on the tape.
    - The tape density (DENSITY attribute). This value must be the same as the density of existing data on the tape to ensure that the new data gets written at

the same density as data already on the tape. Unlike when reading, when you write to a tape, the density is not automatically set for you.

3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =TAPEFILE5^APPEND. It describes the fifth file on a labeled tape. This tape uses IBM labels.

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS          ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to IBM:
ATTRIBUTE^NAME ':=' "LABELS         ";
ATTRIBUTE^VALUE ':=' "IBM" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to MYVOL:
ATTRIBUTE^NAME ':=' "VOLUME         ";
ATTRIBUTE^VALUE ':=' "MYVOL" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to TAPEFILE:
ATTRIBUTE^NAME ':=' "FILEID         ";
ATTRIBUTE^VALUE ':=' "TAPEFILE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 5:
ATTRIBUTE^NAME ':=' "FILESEQ        ";
ATTRIBUTE^VALUE ':=' "5";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the USE attribute to EXTEND:
ATTRIBUTE^NAME ':=' "USE             ";
ATTRIBUTE^VALUE ':=' "EXTEND" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the EBCDIC attribute to OUT:
ATTRIBUTE^NAME ':=' "EBCDIC          ";
ATTRIBUTE^VALUE ':=' "OUT" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECFORM attribute to F:
ATTRIBUTE^NAME ':=' "RECFORM         ";
ATTRIBUTE^VALUE ':=' "F";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECLEN attribute to 512:
ATTRIBUTE^NAME ':=' "RECLEN          ";
ATTRIBUTE^VALUE ':=' "512" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the BLOCKLEN attribute to 2048:
ATTRIBUTE^NAME ':=' "BLOCKLEN        ";
ATTRIBUTE^VALUE ':=' "2048" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the DENSITY attribute to 1600:
ATTRIBUTE^NAME ':=' "DENSITY         ";
ATTRIBUTE^VALUE ':=' "1600" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...
```

```
!Create the DEFINE:
DEFINE^NAME ':=' "=TAPEFILE5^APPEND       ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Writing to the File

Use the DEFINE created above for appending to the file as described below. Note that you can append only to the last file on the tape. So, in this case, the fifth file must also be the last file.

1. Open the DEFINE using the FILE_OPEN_ procedure. If the file exists and the DEFINE attributes match those on the tape label, then the file is opened. If the file does not exist, it is created and opened; the file sequence number must be one greater than that of the last file on the tape.

   The returned file number is related to the tape drive that the tape is mounted on.

2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Write records to the file using the WRITE[X] procedure.

The following code fragment writes to the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to block four records into one record block before the record block is written to tape in one write operation.

```
LITERAL BUFFERED^MODE = 99,
        ON = 1;
 .
 .
!Open the tape file:
FILE^NAME ':=' "=TAPEFILE5^APPEND" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ON);
IF <> THEN ...
 .
 .

!Block four records into the output buffer:
SBUFFER[0]    ':=' LOGICAL^BUFFER^1[0] FOR 512;
SBUFFER[512]  ':=' LOGICAL^BUFFER^2[0] FOR 512;
SBUFFER[1024] ':=' LOGICAL^BUFFER^3[0] FOR 512;
SBUFFER[1536] ':=' LOGICAL^BUFFER^4[0] FOR 512 -> @S^PTR;

!Write a record block to the tape file:
CALL WRITEX(TAPE^NUM,
            SBUFFER,
            @S^PTR '-' @SBUFFER);
```

.
.

Writing to a File on Multiple Labeled Tape Volumes

The procedure for writing to a file that resides on multiple labeled tapes is similar to the procedure for writing to a file on a single tape reel. Again you use a DEFINE to describe the file and the type of operation you intend to perform. Then you open and write to the DEFINE.

The following paragraphs show the complete procedure for writing records to a file on multiple reels of labeled tape.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.
2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
    - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”
    - The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
    - The volume identifier (VOLUME attribute). This value should specify a list of volume names starting with the first volume where the file resides.
    - The file identifier (FILEID attribute). If the file already exists, the file identifier must be the same as the file identifier in the file label. If the file does not yet exist, the file identifier can be any valid file identifier.
    - The file sequence number (FILESEQ attribute). This value must be set to 1 (or not specified, as the default is 1).
    - The I/O operation (USE attribute). This value must be set to “EXTEND” to append records to the file or “OUT” to write to a new file section.
    - The conversion mode (EBCDIC attribute). Set this value to “OUT” to convert ASCII code to EBCDIC on output. Use this option only if the tape uses IBM standard labels.
    - The record type (RECFORM attribute). Specify fixed-length or variable-length records, as appropriate.
    - The record length (RECLEN attribute). If you are appending to an existing file, this value must be equal to the value written to the file label when the tape file was created.
    - The record-block length (BLOCKLEN attribute). Set this value to either a multiple of the fixed record length or the maximum variable record length. If you are creating a new file, this value is placed in the tape label. If you are

appending to a file, this value must also equal the corresponding value in the file label on the tape.

- The tape density (DENSITY attribute). This value must be the same as the density of existing data on the tape to ensure that the new data gets written to tape at the same density as data already on the tape. Unlike when reading, when you write to a magnetic tape, the density is not automatically set to the density of data already on the tape.

3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =MY^TAPE^UPDATE. When writing to the file described by this DEFINE, the system prompts the user to mount a new tape when the end of a tape is reached. This is done transparently to the application program. The tape uses standard ANSI labels.

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS          ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to ANSI:
ATTRIBUTE^NAME ':=' "LABELS         ";
ATTRIBUTE^VALUE ':=' "ANSI" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to MYVOL:
ATTRIBUTE^NAME ':=' "VOLUME         ";
ATTRIBUTE^VALUE ':=' "MYVOL" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to 1_TAPEFILE:
ATTRIBUTE^NAME ':=' "FILEID         ";
ATTRIBUTE^VALUE ':=' "1_TAPEFILE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 1:
ATTRIBUTE^NAME ':=' "FILESEQ         ";
ATTRIBUTE^VALUE ':=' "1";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the USE attribute to OUT:
ATTRIBUTE^NAME ':=' "USE             ";
ATTRIBUTE^VALUE ':=' "OUT" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECFORM attribute to F:
ATTRIBUTE^NAME ':=' "RECFORM         ";
ATTRIBUTE^VALUE ':=' "F";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the RECLEN attribute to 512:
ATTRIBUTE^NAME ':=' "RECLEN          ";
ATTRIBUTE^VALUE ':=' "512" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the BLOCKLEN attribute to 2048:
ATTRIBUTE^NAME ':=' "BLOCKLEN        ";
ATTRIBUTE^VALUE ':=' "2048" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the DENSITY attribute to 1600:
ATTRIBUTE^NAME ':=' "DENSITY         ";
ATTRIBUTE^VALUE ':=' "1600" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=MY^TAPE^UPDATE         ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Writing to the File

Use the DEFINE created in the previous example for writing to the file as described below. Note that the DEFINE refers to the last tape reel of a four-tape file. You can write or append only to the last tape in the file.

1. Open the DEFINE using the FILE_OPEN_ procedure. If the file section exists and the DEFINE attributes match those on the tape label, then the file is opened.

   The returned file number refers to the tape drive that the tape is mounted on.

2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Write records to the file using the WRITE[X] procedure.

The following code fragment updates records on the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to block four records into one record block before the record block is written to tape in one write operation.

```
LITERAL SPACE^FORWARD = 9,
        BUFFERED^MODE = 99,
        ON = 1;
 .
 .

!Open the tape file:
FILE^NAME ':=' "=MY^TAPE^UPDATE" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,ON);
IF <> THEN ...
 .
 .

!Block four records into the output buffer:
SBUFFER[0] ':=' LOGICAL^BUFFER^1[0] FOR 512 -> @S^PTR;
S^PTR ':=' LOGICAL^BUFFER^2[0] FOR 512 -> @S^PTR;
S^PTR ':=' LOGICAL^BUFFER^3[0] FOR 512 -> @S^PTR;
S^PTR ':=' LOGICAL^BUFFER^4[0] FOR 512 -> @S^PTR;

!Write a record block to the tape file:
CALL WRITEX(TAPE^NUM,SBUFFER,
            @S^PTR '-' @SBUFFER);
```

```
  .
  .
```

Reading From the Only File on a Labeled Tape Volume

Like writing, reading from the only file on a labeled tape volume requires a DEFINE that accurately describes the file and the operation you intend to perform on the file. Then you read from the file identified by the created DEFINE.

A DEFINE for reading differs from a DEFINE made for writing. You must set the USE attribute to “IN.” If the tape is an IBM file, you need to set the EBCDIC attribute to “IN.” Also, you must specify the VOLUME attribute. (The VOLUME attribute is optional when the USE attribute is set to “OUT.”)

Note that when reading from a tape, it is not necessary to specify the tape density. The tape controller can determine the correct density.

Just as with writing to the file, you create the DEFINE once and then use it whenever you want to read from the file.

The following paragraphs describe how to create a DEFINE for this type of tape access and how to read from the file once the DEFINE exists.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.
2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
   - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”
   - The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
   - The volume identifier (VOLUME attribute). Set this value to the value written to the tape in the volume label.
   - The file identifier (FILEID attribute). Set this value to the value written to the tape in the file label.
   - The file sequence number (FILESEQ attribute). This value must be set to 1 (or not specified, as the default is 1) because it is the first file on the tape.
   - The I/O operation (USE attribute). This value must be set to “IN” to read records from the file.
   - The conversion mode (EBCDIC attribute). Set this value to “IN” to convert EBCDIC code to ASCII on input. Use this option only if the tape uses IBM standard labels.
3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =TAPEFILE^READ that describes a labeled tape file using standard ANSI labels:

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS           ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to ANSI:
ATTRIBUTE^NAME ':=' "LABELS          ";
ATTRIBUTE^VALUE ':=' "ANSI" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to MYVOL:
ATTRIBUTE^NAME ':=' "VOLUME          ";
ATTRIBUTE^VALUE ':=' "MYVOL" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to TAPEFILE:
ATTRIBUTE^NAME ':=' "FILEID          ";
ATTRIBUTE^VALUE ':=' "TAPEFILE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 1:
ATTRIBUTE^NAME ':=' "FILESEQ         ";
ATTRIBUTE^VALUE ':=' "1";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the USE attribute to IN:
ATTRIBUTE^NAME ':=' "USE             ";
```

```
ATTRIBUTE^VALUE ':=' "IN" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=TAPEFILE^READ          ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Reading From the File

Use the DEFINE created above for reading from the file as follows:

1. Open the DEFINE using the FILE_OPEN_ procedure. If the DEFINE attributes match the attributes in the tape label, then the file is opened. The returned file number refers to the tape drive that the tape is mounted on.
2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Read records from the file using the READ[X] procedure.

The following code fragment reads from the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to deblock each record block into four records before the application can make use of the returned record block.

```
LITERAL SPACE^FORWARD = 9,
        BUFFERED^MODE = 99,
        ON = 1;
 .
 .

!Open the tape file:
FILE^NAME ':=' "=TAPEFILE^READ" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,BUFFERED^MODE,ON);
IF <> THEN ...
 .
 .

!Position the tape to the desired record block:
PHYSICAL^RECORD^ADVANCE := 36;
CALL CONTROL(TAPE^NUM,SPACE^FORWARD,PHYSICAL^RECORD^ADVANCE);

!Read a record block from the tape file into the input
!buffer:
RCOUNT := 2048;
```

```
CALL READX(TAPE^NUM,SBUFFER,
           RCOUNT,COUNT^READ);

!Deblock the input buffer into four records:
LOGICAL^BUFFER^1[0] ':=' SBUFFER[0] FOR 512;
LOGICAL^BUFFER^2[0] ':=' SBUFFER[512] FOR 512;
LOGICAL^BUFFER^3[0] ':=' SBUFFER[1024] FOR 512;
LOGICAL^BUFFER^4[0] ':=' SBUFFER[1536] FOR 512;
 .
 .
```

Reading From a File on a Multiple-File Labeled Tape Volume

If the labeled tape contains multiple files, the procedure for reading records from the file is similar to that for reading from the only file on a labeled tape volume; you need to create a DEFINE and then use the DEFINE for reading from the file. The difference is that the FILESEQ attribute specified in the DEFINE must identify the correct file.

Again, it is not necessary to specify the tape density when reading. The tape controller can calculate the density by reading the tape.

The following paragraphs show how to create a DEFINE for this type of access and then use the DEFINE to read records from a file that resides on a labeled tape containing other files.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.
2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
    - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”
    - The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
    - The volume identifier (VOLUME attribute). Set this value to the value written to the tape in the volume label.
    - The file identifier (FILEID attribute). The file identifier must be the same as the file identifier in the file label.
    - The file sequence number (FILESEQ attribute). This value must be set to the sequence number of the file on the tape. For example, if the DEFINE will describe the seventh file on the tape, then this attribute must be set to 7.
    - The I/O operation (USE attribute). This value must be set to “IN” to read records from the file.
    - The conversion mode (EBCDIC attribute). Set this value to “IN” to convert EBCDIC code to ASCII on input. Use this option only if the tape uses IBM standard labels.

3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =FILE^FIVE^READ. It describes the fifth file on a labeled tape. This tape uses IBM labels.

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS          ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to IBM:
ATTRIBUTE^NAME ':=' "LABELS         ";
ATTRIBUTE^VALUE ':=' "IBM" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to MYVOL:
ATTRIBUTE^NAME ':=' "VOLUME         ";
ATTRIBUTE^VALUE ':=' "MYVOL" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to FILE^FIVE:
ATTRIBUTE^NAME ':=' "FILEID         ";
ATTRIBUTE^VALUE ':=' "FILE^FIVE" -> @S^PTR;

ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 5:
ATTRIBUTE^NAME ':=' "FILESEQ        ";
ATTRIBUTE^VALUE ':=' "5";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the USE attribute to IN:
ATTRIBUTE^NAME ':=' "USE             ";
ATTRIBUTE^VALUE ':=' "IN" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the EBCDIC attribute to IN:
ATTRIBUTE^NAME ':=' "EBCDIC          ";
ATTRIBUTE^VALUE ':=' "IN" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=FILE^FIVE^READ         ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Reading From the File

Use the DEFINE created above for reading from the file as described below.

1. Open the DEFINE using the FILE_OPEN_ procedure. If the file exists and the DEFINE attributes match those on the tape label, then the file is opened. The returned file number relates to the tape drive that the tape is mounted on.
2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Read records from the file using the READ[X] procedure.

The following code fragment reads from the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to separate each record block into four records before the application can make use of the data.

```
LITERAL SPACE^FORWARD = 9,
        BUFFERED^MODE = 99,
        ON = 1;
 .
 .

!Open the tape file:
FILE^NAME ':=' "=FILE^FIVE^READ" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ON);
IF <> THEN ...
 .
 .

!Position the tape to the desired record block:
PHYSICAL^RECORD^ADVANCE := 36;
CALL CONTROL(TAPE^NUM,
             SPACE^FORWARD,
             PHYSICAL^RECORD^ADVANCE);

!Read a record block from the tape file into the input
!buffer:
RCOUNT := 2048;
CALL READ(TAPE^NUM,
          SBUFFER,
          RCOUNT,
          COUNT^READ);

!Deblock the input buffer into four records:
LOGICAL^BUFFER^1[0] ':=' SBUFFER[0] FOR 512;
LOGICAL^BUFFER^2[0] ':=' SBUFFER[512] FOR 512;
LOGICAL^BUFFER^3[0] ':=' SBUFFER[1024] FOR 512;
LOGICAL^BUFFER^4[0] ':=' SBUFFER[1536] FOR 512;
 .
 .
```

Reading From a File on Multiple Labeled Tape Volumes

The procedure for reading from a file that resides on multiple labeled tapes is similar to the procedure for reading from a file on a single tape reel. Again you describe the file and the type of operation you intend to perform in a DEFINE, open the DEFINE, and read from the file associated with the returned file number.

The following paragraphs show how to create a DEFINE for this type of tape access, then how to use the DEFINE for reading record blocks from a file on multiple reels of labeled tape.

Creating the DEFINE

Create the DEFINE as follows:

1. Turn on DEFINEs by calling the DEFINEMODE procedure.
2. Create a working set for the DEFINE using successive calls to the DEFINESETATTR procedure. The working set should include the following:
    - The class of DEFINE (CLASS attribute). Set this value to “TAPE.”

- The type of labels used (LABELS attribute). Set this value to “ANSI” or “IBM.”
- The volume identifier (VOLUME attribute). This value should specify a list of volume names starting with the first volume where the file resides.
- The file identifier (FILEID attribute). The file identifier must be the same as the file identifier in the file label.
- The file sequence number (FILESEQ attribute). This value must be set to 1.
- The I/O operation (USE attribute). This value must be set to “IN” to read records from the file.
- The conversion mode (EBCDIC attribute). Set this value to “IN” to convert EBCDIC code to ASCII on input. Use this option only if the tape uses IBM standard labels.

3. Create the DEFINE using the DEFINEADD procedure.

The following example creates a DEFINE called =THIRD^TAPE^READ. It describes the third section of the file: that part contained on the third tape reel. This tape uses standard ANSI labels.

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS          ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to ANSI:
ATTRIBUTE^NAME ':=' "LABELS         ";
ATTRIBUTE^VALUE ':=' "ANSI" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the VOLUME attribute to THIRD:
ATTRIBUTE^NAME ':=' "VOLUME         ";
ATTRIBUTE^VALUE ':=' "THIRD" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILEID attribute to 4_TAPEFILE:
ATTRIBUTE^NAME ':=' "FILEID          ";
ATTRIBUTE^VALUE ':=' "4_TAPEFILE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESEQ attribute to 1:
ATTRIBUTE^NAME ':=' "FILESEQ         ";
ATTRIBUTE^VALUE ':=' "1";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the REELS attribute to 4:
ATTRIBUTE^NAME ':=' "REELS           ";
ATTRIBUTE^VALUE ':=' "4";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the FILESECT attribute to 3:
ATTRIBUTE^NAME ':=' "FILESECT        ";
ATTRIBUTE^VALUE ':=' "3";
ATTRIBUTE^LEN := 1;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       ATTRIBUTE^LEN,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the USE attribute to IN:
ATTRIBUTE^NAME ':=' "USE             ";
ATTRIBUTE^VALUE ':=' "IN" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=THIRD^TAPE^READ         ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

Reading From the File

Use the DEFINE created above for reading the file as described below. Note that the DEFINE refers to the third tape of a four-tape file.

1. Open the DEFINE using the FILE_OPEN_ procedure. If the file section exists and the DEFINE attributes match those on the tape label, then the tape file is opened. The returned file number refers to the tape drive that the tape is mounted on.
2. Turn on buffered mode, if desired, using SETMODE function 99.
3. Read record blocks from the file using the READ[X] procedure.

The following code fragment reads record blocks from the tape using the DEFINE created above. Note that because the record block is four times the size of the record, the application needs to deblock each record block into four records.

```
LITERAL SPACE^FORWARD = 9,
        BUFFERED^MODE = 99,
        ON = 1;
  .
  .
!Open the tape file:
FILE^NAME ':=' "=THIRD^TAPE^READ" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    TAPE^NUM);
IF ERROR <> 0 THEN ...

!Set buffered mode:
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ON);
IF <> THEN ...
  .
  .
!Position the tape to the desired record block:
PHYSICAL^RECORD^ADVANCE := 36;
CALL CONTROL(TAPE^NUM,
             SPACE^FORWARD,
             PHYSICAL^RECORD^ADVANCE);

!Read a record block from the tape file into the input
!buffer:
RCOUNT := 2048;
CALL READ(TAPE^NUM,
          SBUFFER,
          RCOUNT,COUNT^READ);

!Deblock the input buffer into four records:
LOGICAL^BUFFER^1[0] ':=' SBUFFER[0] FOR 512;
LOGICAL^BUFFER^2[0] ':=' SBUFFER[512] FOR 512;
```

```
LOGICAL^BUFFER^3[0] ':=' SBUFFER[1024] FOR 512;
LOGICAL^BUFFER^4[0] ':=' SBUFFER[1536] FOR 512;

  .
  .
```

Accessing a Labeled Tape File: An Example

In this subsection, the program used in Section 5, Communicating With Disk Files, for saving a daily log in an entry-sequenced disk file is modified. Now, because of the sequential nature of the application, this example will be used to show communication with magnetic tape.

This example uses a labeled tape file. The program itself works with standard ANSI labels or standard IBM labels, so long as the label type specified in the DEFINE matches the label type on the tape.

Preparing the Tape

Before running the program, the tape must have been prepared with appropriate tape labels. A user with super-group user privilege must apply the labels to the tape using the ADD TAPELABEL command of the MEDIACOM utility or the LABEL command of the TAPECOM utility. For example, using the MEDIACOM utility:

```
1> MEDIACOM
MEDIACOM - T6028D20 (01JUN93)
Copyright Tandem Computers Incorporated 1993
MC>add tapelabel myvol,tapedrive $tape,unload off,override on
TAPE VOLUME MYVOL INITIALIZED

MC>exit
```

Using the TAPECOM utility:

```
1> TAPECOM
TAPECOM - T6985D00 (12DEC91) GUARDIAN 90
Copyright Tandem Computers Incorporated 1985-91
?label myvol, device $tape, nounload
STATUS 2501 - VOLUME MYVOL INITIALIZED

?exit
```

Refer to the *DSM/TC Operator Interface (MEDIACOM)* for details on the MEDIACOM utility; refer to the *Guardian Disk and Tape Utilities Reference Manual* for details on the TAPECOM utility.

Creating the DEFINE

The example given below uses a record-block size of 2048 bytes and a record size of 512 bytes. Each DEFINE that accesses the file has its BLOCKLEN attribute set to 2048, its RECLEN attribute set to 512, and its RECFORM attribute set to “F.” In addition, the VOLUME and FILEID attributes must match those on the tape, and the

FILESEQ attribute must be set to 1 because the file is the first and only file on the tape. Therefore three DEFINEs are created as shown below.

The program opens the following DEFINE for reading from the tape:

```
1> SET DEFINE CLASS TAPE, LABELS ANSI, USE IN, VOLUME MYVOL,
   FILESEQ 1,FILEID FILE1
```

```
2> ADD DEFINE =READ^TAPE
```

For appending to the tape:

```
3> SET DEFINE CLASS TAPE, LABELS ANSI, USE EXTEND,
   VOLUME MYVOL, FILESEQ 1, FILEID FILE1, BLOCKLEN 2048,
   RECLEN 512, RECFORM F
```

```
4> ADD DEFINE =APPEND^TAPE
```

For creating the file and writing to the new file:

```
5> SET DEFINE CLASS TAPE, LABELS ANSI, USE OUT, VOLUME MYVOL,
   FILESEQ 1, FILEID FILE1, BLOCKLEN 2048, RECLEN 512,
   RECFORM F
```

```
6> ADD DEFINE =CREATE^FILE
```

Writing the Program

The sample program allows the user to read records from the file, append records to the file, or create the file and write records to it. The program consists of the following procedures:

- The LOGGER procedure is the MAIN procedure. It calls GET^COMMAND to prompt the user to select a function (read, append, create, or exit), and then calls the appropriate procedure.
- The INIT and SAVE^STARTUP^MESSAGE procedures save the Startup message in the global data area and open the process IN file for terminal I/O. In addition, the INIT procedure checks that labeled tape support has been turned on.
- The READ^RECORD procedure opens the file for reading and prompts the user for a record number. The procedure calculates the record-block number by dividing the record number by 4 and then reads the corresponding record block. Using modulo division, the procedure calculates which of the four records contained in the record block is required, and then it prints the date and commentary text on the terminal.

  After printing out the record, the procedure prompts the user to read the next record. If the user declines, then the procedure returns control to LOGGER. Otherwise, the program displays the next record. If the next record is part of a different record block, then the procedure reads in the next record block from tape.
- The APPEND^RECORD procedure opens the file for appending. Once the tape file is open, the procedure prompts the user to enter the date and commentary text. The procedure then prompts the user to enter another record. If the user declines,

the record is put into the tape buffer and written to tape as a partial record block. If the user chooses to enter more records, the procedure blocks each record into the tape buffer until either the buffer contains four records or the user declines to enter more records. At this point the procedure writes the tape buffer to tape—one record block.

- The OPEN^TAPE^FILE procedure is called from either the READ^RECORD or APPEND^RECORD procedures to open the tape file using the CLASS TAPE DEFINE appropriate for the selected function. If append is selected, this procedure uses the =APPEND^TAPE DEFINE. If create is selected, this procedure uses the =CREATE^TAPE DEFINE, which is like the =APPEND^TAPE DEFINE except that it also writes new file labels to the tape. If read is selected, this procedure uses the =READ^TAPE DEFINE.

  This procedure also sets buffered mode for the tape file.

- The FILE^ERRORS and FILE^ERRORS^NAME procedures respond to file system errors. They simply print out the error number and stop the program.
- Procedure ILLEGAL^COMMAND informs the user of an invalid command selection and then returns to the LOGGER procedure to prompt for another function.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

LITERAL  BUFSIZE  = 512;
LITERAL  TBUFSIZE = 2048;
LITERAL  MAXFLEN  = ZSYS^VAL^LEN^FILENAME;
LITERAL  ABEND    = 1;

STRING  .SBUFFER[0:BUFSIZE];        !Buffer for terminal I/O
INT      TERMNUM;                   !Terminal file number
INT      TAPENUM;                   !Tape file number
STRING   CMD;                       !Function executing
STRING  .S^PTR;                     !String pointer

INT     .TBUFFER[0:(TBUFSIZE/2 - 1)]; !Buffer for tape I/O
INT     .LREC0 := @TBUFFER[0];      !Integer pointers to
INT     .LREC1 := @TBUFFER[256];    ! records in tape
INT     .LREC2 := @TBUFFER[512];    ! buffer
INT     .LREC3 := @TBUFFER[768];

INT      INDEX;                     !Index into record block
INT(32)  RBLOCK;                    !Record block number

STRUCT  .LOG^RECORD;                !Record structure
BEGIN
   STRING   DATE[0:7];
   STRING   COMMENTS[0:503];
END;
INT     .RECORD^POINTER := @LOG^RECORD[0];

STRUCT  .CI^STARTUP;                !Startup message
```

```
BEGIN
   INT   MSGCODE;
   STRUCT   DEFAULT;
   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
   END;
   STRUCT   INFILE;
   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
      INT   FILEID[0:3];
   END;

   STRUCT   OUTFILE;
   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
      INT   FILEID[0:3];
   END;
   STRING   PARAM[0:529];
END;

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?   FILE_OPEN_,WRITEREADX,WRITEX,PROCESS_STOP_,READX,CONTROL,
?   DNUMOUT,FILE_GETINFO_,DNUMIN,SETMODE,LABELEDTAPESUPPORT,
?   OLDFILENAME_TO_FILENAME_,FILE_CLOSE_)
?LIST


!-------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and print
! messages.
!-------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N) =
          @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE PRINT^LINE =
          CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE PRINT^BLANK =
          CALL WRITE^LINE(SBUFFER,0)   #;

  !  Print a string:

     DEFINE PRINT^STR(S) = BEGIN START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;  END  #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there is no file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING   .FNAME;
INT       LEN;
INT       ERROR;

BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_(!process^handle!,
                      !specifier!,
                      ABEND);
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and the error number are determined from the file
! number and FILE^ERRORS^NAME is then called to do the
! display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT          FNUM;

BEGIN
   INT       ERROR;
   STRING   .FNAME[0:MAXFLEN - 1];
   INT       FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! Procedure to write a message on the terminal and check for
! any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to open a labeled tape file by opening the
! appropriate CLASS TAPE DEFINE.
!------------------------------------------------------------

PROC OPEN^TAPE^FILE;
BEGIN
   INT     ERROR;
   STRING .TAPE^NAME[0:MAXFLEN - 1];

!  Select the CLASS TAPE DEFINE for the requested function:

   CASE CMD OF
   BEGIN
      "a", "A" -> TAPE^NAME ':=' "=APPEND^TAPE" -> @S^PTR;
      "c", "C" -> TAPE^NAME ':=' "=CREATE^TAPE" -> @S^PTR;
      "r", "R" -> TAPE^NAME ':=' "=READ^TAPE" -> @S^PTR;
      OTHERWISE -> ;
   END;

!  Open the selected DEFINE with exclusive mode:

   ERROR :=  FILE_OPEN_(
             TAPE^NAME:@S^PTR '-' @TAPE^NAME,TAPENUM);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(
           TAPE^NAME:@S^PTR '-' @TAPE^NAME,ERROR);

!  Set buffered mode:

   CALL SETMODE(TAPENUM,99,1);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
END;
```

```
!------------------------------------------------------------
! This procedure executes when you press "r" in response to
! the function prompt in the main procedure. It prompts the
! user for the desired record, displays it on the terminal,
! then prompts for sequential reads.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT          COUNT^READ;
   INT(32)      RECORD^NUM;
   STRING .EXT  NEXT^ADDR;
   INT          STATUS;
   INT          ERROR;

!  Open the tape DEFINE and set buffered mode:

   CALL OPEN^TAPE^FILE;

!  Prompt the user to select a record:

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Record Number:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   SBUFFER[COUNT^READ] := 0;

!  Convert ASCII to numeric:

   @NEXT^ADDR := DNUMIN(SBUFFER,RECORD^NUM,10,STATUS);
   IF STATUS OR @NEXT^ADDR <> $XADR(SBUFFER[COUNT^READ])
   THEN
   BEGIN
      PRINT^STR("Error in the record number");
      GOTO PROMPT^AGAIN;
   END;

!  Calculate record block number, assuming blocking
!  factor of 4:

   RBLOCK := RECORD^NUM / 4D;

!  Modulo divide to get record index:

   INDEX := RECORD^NUM '\' 4;

!  Space tape forward to start of record block:

   CALL CONTROL(TAPENUM,9,$INT(RBLOCK));
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
```

```
!  Execute loop if reading just selected, or user
!  has requested to read an additional record.
!  Exit loop if user declines to read next record:

   DO BEGIN

      PRINT^BLANK;

   !  Read a record block from the tape file:

      CALL READX(TAPENUM,TBUFFER,TBUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TAPENUM,ERROR);
         IF ERROR = 1 THEN
         BEGIN
            PRINT^STR("No such record. ");
            RETURN;
         END
         ELSE CALL FILE^ERRORS(TAPENUM);
      END;

      DO BEGIN

      !  Extract the record:

         CASE INDEX OF
         BEGIN
            0 -> LOG^RECORD[0] ':=' LREC0[0] FOR 256;
            1 -> LOG^RECORD[0] ':=' LREC1[0] FOR 256;
            2 -> LOG^RECORD[0] ':=' LREC2[0] FOR 256;
            3 -> LOG^RECORD[0] ':=' LREC3[0] FOR 256;
            OTHERWISE -> CALL PROCESS_STOP_(!process^handle!,
                                            !specifier!,
                                            ABEND);
         END;

      !  Check for incomplete record block.  If this record
      !  is blank, set INDEX to 4 in preparation for reading
      !  the next record block:

         IF LOG^RECORD.DATE = "      " THEN
         BEGIN
            INDEX := 4;
            SBUFFER[0] ':=' "Y";
         END
         ELSE

      !  Process the log record:

         BEGIN

         !  Display date from the record:

            CALL WRITEX(TERMNUM,LOG^RECORD.DATE,8);
            IF <> THEN CALL FILE^ERRORS(TERMNUM);

         !  Display comments:

            CALL WRITEX(TERMNUM,LOG^RECORD.COMMENTS,504);
            IF <> THEN CALL FILE^ERRORS(TERMNUM);

         !  Increment record counter:

            INDEX := INDEX + 1;

         !  Prompt the user to read the next record:

            PRINT^BLANK;
            SBUFFER ':='
               "Do You Want To Read the Next Record (y/n)  "
               -> @S^PTR;
            CALL WRITEREADX(TERMNUM,SBUFFER,
                            @S^PTR '-' @SBUFFER,
                            BUFSIZE,COUNT^READ);
            IF <> THEN CALL FILE^ERRORS(ERROR);
         END;
      END
      UNTIL (NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y"))
             OR INDEX = 4;

   !  No more records in this record block.  Reset record
   !  count to 0 and read next record, if requested:

      INDEX := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");

!  Close the tape file:

   CALL FILE_CLOSE_(TAPENUM);
END;
```

```
!-------------------------------------------------------------
! Procedure to append a record to the file
!-------------------------------------------------------------

PROC APPEND^RECORD;
BEGIN
   INT   ERROR;
   INT   COUNT^READ;
   INT   SEQ^NUM := 0;

!  Open the tape file and set buffered mode:

   CALL OPEN^TAPE^FILE;

!  Blank tape buffer:

   TBUFFER[0] ':=' "  ";
   TBUFFER[1] ':=' TBUFFER[0] FOR 1023;

!  Initialize the index into the tape buffer:

   INDEX := 0;

!  Write records to file. This loop prompts the user for
!  each additional record to be written:

   DO BEGIN

   !  Blank the log record structure:

      RECORD^POINTER[0] ':=' "  ";
      RECORD^POINTER[1] ':=' RECORD^POINTER[0] FOR 255;

   !  Prompt user for date:

   PROMPT^AGAIN:
      PRINT^BLANK;
      SBUFFER ':=' "Enter Today's Date (mmddyyyy):  "
                   -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      IF COUNT^READ <> 8 THEN GOTO PROMPT^AGAIN;

   !  Put date into record structure:

      LOG^RECORD.DATE ':=' SBUFFER[0] FOR COUNT^READ;

   !  Prompt user for comments:

      SBUFFER ':=' "Please Enter Your Comments: " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
```

```
!  Put comments into record structure:

   LOG^RECORD.COMMENTS ':=' SBUFFER[0] FOR COUNT^READ;

!  Pack record into record block:

   CASE INDEX OF
   BEGIN
      0 -> LREC0 ':=' LOG^RECORD FOR 256;
      1 -> LREC1 ':=' LOG^RECORD FOR 256;
      2 -> LREC2 ':=' LOG^RECORD FOR 256;
      3 -> LREC3 ':=' LOG^RECORD FOR 256;
      OTHERWISE -> CALL PROCESS_STOP_;
   END;

!  Prompt the user to enter additional records:

   PRINT^BLANK;
   SBUFFER ':='
           "Do You Wish to Enter Another Record (y/n)? "
           -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Increment the index into the record block:

   INDEX := INDEX + 1;

!  Send record block to tape process if no more records,
!  or if record block full. Flush out to tape every 10
!  writes to provide known point of consistency:

   IF INDEX = 4 OR
      (NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y")) THEN
   BEGIN
      CALL WRITEX(TAPENUM,TBUFFER,TBUFSIZE);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TAPENUM,ERROR);
         IF ERROR <> 1 THEN CALL FILE^ERRORS(TAPENUM);
      END;

   !  Increment the record block count and reset the
   !  index:

      SEQ^NUM := SEQ^NUM + 1;
      INDEX := 0;

   !  Blank tape buffer in case next record block is not
   !  full:

      TBUFFER[0] ':=' "  ";
      TBUFFER[1] ':=' TBUFFER[0] FOR 1023;
      !  Flush to tape every 10 record blocks. Use modulo
      !  divide to detect tenth record. Buffered mode is
      !  already set, therefore SETMODE 99 forces to tape all
      !  records in tape buffer:

         IF $DBL(SEQ^NUM) '\' 10 = 0 THEN
         BEGIN
            CALL SETMODE(TAPENUM,99,1);
            IF <> THEN CALL FILE^ERRORS(TAPENUM);
         END;
      END;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");

!  Turn off buffered mode:

   CALL SETMODE(TAPENUM,99,0);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

!  Close the tape file:

   CALL FILE_CLOSE_(TAPENUM);

END;


!-------------------------------------------------------------
! Procedure to stop the program on request. As well as
! stopping the program, this procedure rewinds and unloads
! the tape.
!-------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN

!  Stop the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure to process an illegal command.  The procedure
! informs the user that the selection was other than "r,"
! "a," "c," or "x."
!------------------------------------------------------------

PROC ILLEGAL^COMMAND;

!  If user selects other than r, a, c, or x:
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid
!  then return to prompt again for a valid function:

   PRINT^STR("ILLEGAL COMMAND: " &
             "Type one of 'r,' 'a,' 'c,' or 'x.'");
END;


!------------------------------------------------------------
! Procedure to prompt the user for the next function to be
! performed:
!
!  "r" to read records
!  "a" to append records
!  "c" to create a file and append records
!  "x" to exit the program
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'a' for Append to Log, ");
   PRINT^STR("     'c' for Create File and Append, ");
   PRINT^STR("     'x' for Exit. ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to save Startup message in a global structure.
!------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH, MATCH) VARIABLE;

INT    .RUCB;
INT    .START^DATA;
INT    .MESSAGE;
INT     LENGTH;
INT     MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;
```

```
!------------------------------------------------------------
! Procedure to perform initialization for the program.  It
! calls INITIALIZER to read and copy the Startup message
! into the global variables area and then opens the IN file
! specified in the Startup message.  This procedure also
! checks whether labeled tape support is turned on.
!------------------------------------------------------------

PROC INIT;
BEGIN
   INT     OPEN^FLAG;
   INT     ERROR;
   INT     RETURNED^VALUE;
   INT    .TERM^NAME[0:MAXFLEN - 1];
   INT     TERMLEN;

!  Read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthrough!,
                    SAVE^STARTUP^MESSAGE);

!  Open IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
            CI^STARTUP.INFILE.VOLUME,
            TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERMNUM);
   IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                 !specifier!,
                                 ABEND);

!  Check if labeled tape support is turned on.  Print a
!  message and stop the program if not:

   RETURNED^VALUE := LABELEDTAPESUPPORT;
   IF RETURNED^VALUE = 0 THEN
   BEGIN
      PRINT^STR("Labeled tape support is not enabled. ");
      CALL PROCESS_STOP_(!process^handle!,
                         !specifier!,
                         ABEND);
   END;
END;
```

```
!------------------------------------------------------------
! This is the main procedure.
!------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN

   CALL INIT;

!  Loop indefinitely until user selects function x:

   WHILE 1 DO
   BEGIN

   !  Prompts for the function to perform:

      CMD := GET^COMMAND;

   !  Call function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R" -> CALL READ^RECORD;

         "a", "A" -> CALL APPEND^RECORD;

         "c", "C" -> CALL APPEND^RECORD;

         "x", "X" -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL ILLEGAL^COMMAND;
      END;
   END;
END;
```

Working With Unlabeled Tapes

Any tape that does not have standard ANSI, IBM or TMF labels (or for a backup tape either BACKUP or IBMBACKUP labels) is an unlabeled tape. Use the methods described in this subsection for handling either tapes produced by other vendors that don’t have IBM or ANSI labels or tapes produced on HP systems without using standard labeled tape processing.

Accessing Unlabeled Tapes

You gain access to an unlabeled tape by opening the file name of the drive the tape is mounted on. To do this, either:

- Open the tape-drive using a DEFINE, specifying the tape-file name in the DEVICE attribute and setting the LABELS attribute to “OMITTED.”
- Pass the tape-drive name or logical device number to the FILE_OPEN_ procedure.

In either case, you need operator permission to open the file if labeled tape processing is turned on and the operator has set NLCHECK permission using the MEDIACOM utility or the TAPECOM utility. In this case, a message is sent to the operator to enable the operation.

Using the DEFINE method, you can access a labeled tape as if it had no labels by setting the LABELS attribute to “BYPASS.” You need operator permission to open the file if labeled tape processing is turned on and the operator has set BLPCHECK permission using the MEDIACOM utility or the TAPECOM utility. In this case, a message is sent to the operator to enable the operation.

The following example creates a DEFINE for an unlabeled tape mounted on tape drive $TAPE1:

```
!Turn on DEFINE mode:
NEW^VALUE := 1;
ERROR := DEFINEMODE(NEW^VALUE,
                    OLD^VALUE);
IF ERROR > 0 THEN ...

!Set the CLASS attribute to TAPE:
ATTRIBUTE^NAME ':=' "CLASS          ";
ATTRIBUTE^VALUE ':=' "TAPE" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the LABELS attribute to OMITTED:
ATTRIBUTE^NAME ':=' "LABELS         ";
ATTRIBUTE^VALUE ':=' "OMITTED" -> @S^PTR;
```

```
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Set the DEVICE attribute to \SYS2.$TAPE1:
ATTRIBUTE^NAME ':=' "DEVICE         ";
ATTRIBUTE^VALUE ':=' "\SYS2.$TAPE1" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...

!Create the DEFINE:
DEFINE^NAME ':=' "=TAPE1                  ";
ERROR := DEFINEADD(DEFINE^NAME);
IF ERROR <> 0 THEN ...
```

To open the tape file, pass the DEFINE name to the FILE_OPEN_ procedure:

```
FILE^NAME ':=' "=TAPE1" -> @S^PTR;
ERROR :=  FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                     TAPE^NUM);
IF ERROR <> 0 THEN ...
```

Note. The only DEFINE attributes that are allowed for unlabeled tape access are DENSITY, DEVICE, LABELS, and MOUNTMSG. The following attributes are invalid for unlabeled tapes: BLOCKLEN, EBCDIC, EXPIRATION, FILEID, FILESECT, FILESEQ, GEN, OWNER, RECFORM, RECLEN, REELS, RETENTION, SYSTEM, USE, VERSION, VOLUME.

If you open the tape drive without a DEFINE, then you simply pass the name or device number of the tape drive to the FILE_OPEN_ call:

```
FILE^NAME ':=' "\SYS2.TAPE" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@S^PTR '-' @FILE^NAME,
                    FILE^NUM);
IF ERROR <> 0 THEN ...
```

Once the tape drive is open, consider the following before performing read/write operations to the tape:

- Should records be blocked for efficiency?
- If writing to the tape, what should the tape density be?
- What is the device mode and speed?
- Is code conversion necessary?
- Do you intend to use buffered mode?

The following paragraphs discuss the above considerations.

Blocking Tape I/O

There is no support for blocking records that are written to an unlabeled tape. If you choose to do blocking, then your program must pack multiple records into one record block before the record gets written to tape. Similarly, on reading records from the tape, the program must do its own deblocking of record blocks back into records.

Blocking records in this way has the following advantages:

- It speeds up tape I/O because fewer write and read operations are required.
- It uses less tape because there are fewer records and thus fewer interrecord gaps.

Refer to Blocking Tape Records earlier in this section for details.

Specifying Tape Density

When writing to tape, you can specify the density with which you want to write to tape. If you do not set the density, then the system will use the configured default density for the drive. On reading from tape, it is not necessary to specify the density, because the tape controller can calculate the tape density.

If you open the tape using a DEFINE, you can use the same DEFINE to select the tape density by setting the DENSITY attribute. The following code fragment sets the density to 1600 bits per inch:

```
!Set the DENSITY attribute to 1600 bpi:
ATTRIBUTE^NAME ':=' "DENSITY         ";
ATTRIBUTE^VALUE ':=' "1600" -> @S^PTR;
ERROR := DEFINESETATTR(ATTRIBUTE^NAME,
                       ATTRIBUTE^VALUE,
                       @S^PTR '-' @ATTRIBUTE^VALUE,
                       DEFAULT^NAMES);
IF ERROR <> 0 THEN ...
```

The tape density gets set when the tape device is opened using the DEFINE that contains this attribute.

Alternatively, you can set the density using SETMODE function 66. *`parameter-1`* of the SETMODE procedure designates the density, as shown in Table 12-3.

Table 12-3. SETMODE 66 parameter-1 Settings for Tape Density

SETMODE *parameter-1*	Tape Density
1	1600 bpi (PE)
2	6250 bpi (GCR)
3	As indicated by switches on the tape drive (D-series releases only)

The following example also sets the tape density to 1600 bits per inch:

```
LITERAL TAPE^DENSITY = 66;
 .
 .
DENSITY := 1;
CALL SETMODE(TAPE^NUM,
             TAPE^DENSITY,
             DENSITY);
IF <> THEN ...
```

Here, the tape drive must already be open. The selected density becomes effective immediately.

Selecting the Device Mode (D-Series Only)

On D-series releases only, if you are accessing a tape on a drive controlled by a 3209/5120 controller, you can access the cartridge in either start/stop or streaming mode. If you are using this type of drive, you should therefore use SETMODE function 119 to specify the device mode. You set the device mode after opening the tape device but before accessing the tape.

The following example sets the device mode to streaming mode.

```
LITERAL DEVICE^MODE = 119,
        START^STOP  =   0,
        STREAMING   =   1;
 .
 .
CALL SETMODE(TAPE^NUM,
             DEVICE^MODE,
             STREAMING);
IF <> THEN ...
```

A program can be designed to run for any type of tape drive/controller by using the FILE_GETINFO_ procedure immediately after opening the tape file to determine the subdevice type of the file. If the subdevice type is 5 (that is, the 3209 controller for a 5120 device), then you prompt the user for the mode and set the mode accordingly. If the subdevice type is not 5, continue without setting the device mode.

Selecting the Conversion Mode

When reading information from an unlabeled tape written by another vendor’s equipment or writing to an unlabeled tape that will be read by another vendor’s equipment, you must consider whether the tape contains EBCDIC or ASCII code.

For an EBCDIC tape, you need to convert the data on input to ASCII. You can use a FUP command to copy the data from tape to disk or another tape and convert the code to ASCII at the same time. The following example copies an EBCDIC tape from device $TAPE2 to $TAPE1:

```
FUP COPY $TAPE2,$TAPE1,EBCDICIN
```

Similarly, you can convert ASCII code to EBCDIC for output as follows:

```
FUP COPY $TAPE1,$TAPE2,EBCDICOUT
```

Refer to the *File Utility Program (FUP) Reference Manual* for details on the FUP COPY command.

Setting Buffered Mode

As with labeled tape, we recommend using buffered mode to increase throughput. When you use buffered mode for writing to a tape, the application is allowed to continue as soon as the tape process has received the write request. Without buffered mode, the application must wait for each write to tape to finish before continuing. When reading from tape in buffered mode, the tape process reads ahead in anticipation of sequential read operations.

You turn on buffered mode using SETMODE function 99:

```
LITERAL BUFFERED^MODE = 99,
        ON            =  1,
        OFF           =  0;
 .
 .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             ON);
IF <> THEN ...
```

Always turn off buffered mode before closing the tape drive and unloading the tape:

```
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             OFF);
IF <> THEN ...
```

For complete details on buffered mode operation, refer to Working in Buffered Mode, earlier in this section.

Writing to a Single-File Unlabeled Tape

The following paragraphs describe how to write programs to do the following:

- Write a new file to a scratch tape.
- Append to the only file on an unlabeled tape.

Writing a New File to a Scratch Tape

Writing a file to magnetic tape involves two steps:

1. Write records to the tape device using the WRITE procedure; for example:

   ```
   !Write record blocks to tape:
   WHILE NOT DONE
   BEGIN
       .
       .
      !Write one record to tape:
      CALL WRITEX(TAPE^NUM,
                  SBUFFER,
                  WCOUNT);
      IF <> THEN ...
       .
       .

      IF <no more to write> THEN DONE = 1;
   END;
   ```

2. Terminate the file with an end-of-file mark and an indication of end of tape. Because there is only one file on the tape, you could get away without a separate convention for indicating the end of the tape. However, it is worth observing an end-of-tape convention, regardless of how many files the tape contains. In addition to providing consistency between multiple-file and single-file tapes, you will need the end-of-tape convention if you add files to the tape later. Therefore we recommend terminating this file with the HP end-of-tape convention: two end-of-file marks.

   The following code fragment writes two EOF marks to the tape:

   ```
   LITERAL WRITE^EOF = 2;
    .
    .
   !Write EOF mark to signify end of file:
   CALL CONTROL(TAPE^NUM,
                WRITE^EOF);
   IF <> THEN ...

   !Write another end-of-file mark to signify end of tape:
   CALL CONTROL(TAPE^NUM,
                WRITE^EOF);
   IF <> THEN ...
   ```

Appending to the Only File on an Unlabeled Tape

To append to the only file on an unlabeled tape, your program must do the following:

1. Space forward one file. The tape stops immediately after the first EOF mark:

```
LITERAL SPACE^FWD^FILES = 7,
        SPACE^BACK^FILES = 8,
        WRITE^EOF = 2;
 .
 .
NUMBER^OF^FILES := 1;
CALL CONTROL(TAPE^NUM,
             SPACE^FWD^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```

2. Space backward one file. The tape stops immediately before the same EOF mark:

```
NUMBER^OF^FILES := 1;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```

3. Append records to the tape:

```
WHILE NOT DONE
BEGIN
    .
    .
   CALL WRITEX(TAPE^NUM,
               SBUFFER,
               WCOUNT);
   IF <> THEN ...
    .
    .

   IF <no more to write> THEN DONE = 1;
END;
```

4. Write two EOF marks to signify the end of the file and the end of the tape:

```
CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...

CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...
```

Writing to a Multiple-File Unlabeled Tape

Writing records to a multiple-file unlabeled tape reel is similar to writing to a single-file tape reel except that you need to be sure that you write to the appropriate file on the tape. You can add a file to the end of the tape or append records to the last file on the tape. You can find the end of the last file on the tape by searching for two consecutive EOF marks.

Adding Files to the End of a Multiple-File Tape

To append a file to the information already on an unlabeled tape, your program must do the following:

1. Find the double EOF marks that denote the end of information on the tape. One way of doing this is to keep spacing forward one file at a time, reading the first record of each file. If the READ[X] call returns error number 1 (end-of-file warning), then you have reached the end of the tape. The following code fragment positions the tape immediately after the last EOF mark on the tape:

```
LITERAL SPACE^FWD^FILES = 7,
        SPACE^BACK^FILES = 8,
        WRITE^EOF = 2;
 .
 .
WHILE NOT END^OF^TAPE DO
BEGIN

   NUMBER^OF^FILES := 1;
   CALL CONTROL(TAPE^NUM,
                SPACE^FWD^FILES,
                NUMBER^OF^FILES);
   IF <> THEN ...

   READX(TAPE^NUM,
         SBUFFER,
         RCOUNT,
         COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPE^NUM,
                         ERROR);
      IF ERROR = 1 THEN END^OF^TAPE := YES
      ELSE ...            !other error
   END;
END;
```

2. Space backward one EOF mark to position the tape between the two EOF marks:

```
NUMBER^OF^FILES := 1;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```

3. Write records to the tape:

```
WHILE DONE = 0;
BEGIN
     .
     .
CALL WRITEX(TAPE^NUM,
            SBUFFER,
            WCOUNT);
   IF <> THEN ...
     .
     .
   IF <no more to write> THEN DONE := 1;
END;
```

4. Write two EOF marks to the tape to signify the end of the new file and the new end of information on the tape.

```
CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...

CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...
```

Appending Records to a Multiple-File Tape

Appending records to a multiple-file tape is the same as appending to a single-file tape except that you need to space forward to the end of the last file on the tape. That is, you must position the tape before the two EOF marks rather than between them. The following sequence explains how to do this:

1. Find the double EOF marks that denote the end of information on the tape. Again, you can do this by spacing forward one file at a time, reading the first record of each file. If the READ[X] call returns error number 1 (end-of-file warning), then you have reached the end of the tape. The following code fragment positions the tape immediately after the two EOF marks that denote the end of the tape:

```
LITERAL SPACE^FWD^FILES = 7,
        SPACE^BACK^FILES = 8,
        WRITE^EOF = 2;
  .
  .
WHILE NOT END^OF^TAPE DO
BEGIN

   NUMBER^OF^FILES := 1;
   CALL CONTROL(TAPE^NUM,
                SPACE^FWD^FILES,
                NUMBER^OF^FILES);
   IF <> THEN ...
   READX(TAPE^NUM,
         SBUFFER,
         RCOUNT,
         COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPE^NUM,
                         ERROR);
      IF ERROR = 1 THEN END^OF^TAPE := YES
      ELSE ...             !other error
   END;

END;
```

2. Space backward two EOF marks to position the tape at the end of the last file on the tape:

```
NUMBER^OF^FILES := 2;
CALL CONTROL(TAPE^NUM,
             SPACE^BACK^FILES,
             NUMBER^OF^FILES);
IF <> THEN ...
```

3. Write records to the tape:

```
WHILE NOT DONE;
BEGIN
    .
    .
   CALL WRITEX(TAPE^NUM,
               SBUFFER,
               WCOUNT);
   IF <> THEN ...
    .
    .

   IF <no more to write> THEN DONE := 1;
END;
```

4. Write two EOF marks to the tape to signify the new end of the file and the new end of information on the tape:

```
CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...

CALL CONTROL(TAPE^NUM,
             WRITE^EOF);
IF <> THEN ...
```

Writing to a File on Multiple Unlabeled Tape Reels

If your program must deal with files that are too large to fit on a single tape reel, then your program must be able to recognize the end of the tape when writing and to identify the mounted tape reel when reading.

Writing Tape Headers

We recommend using tape headers to identify magnetic tape reels, especially where tape files span multiple tapes. Because we recommend using two EOF marks to denote the end of information on a tape, headers are needed to identify where a multiple-reel file starts and stops. For example, when reading a tape sequentially, your program encounters an EOF mark. The program needs to know whether this is the end of the file or whether the file continues on another tape.

A tape header usually includes information such as an indication as to whether the tape is part of a multiple-tape file, the order of the tape in the file, and the total number of tape reels in the file. It is up to the application designer to choose what information will go into the tape header.

As an alternative to writing your own tape headers, you can use labeled tapes. See Working With Standard Labeled Tapes earlier in this section.

Checking for the End of the Tape

When writing out a multiple-reel file, you need to check for the end of the tape. You do this by checking for the EOT sticker on the tape itself. There may be several records in the buffers that will still be written out to tape after the EOT sticker is encountered. Information can therefore be written beyond the EOT sticker. The program should treat the EOT sticker as a warning that the end of the tape is near and send no more records to the tape process for writing.

The following code fragment writes records to tape. It issues a SETMODE 120, which causes the tape process to return error 150 if the EOT sticker is encountered on a write operation. If EOT is encountered, the program stops writing records and sends two EOF marks to the tape process to indicate the end of information on the tape. Error 150 is expected following each of these write operations and is ignored. Finally, the code fragment issues CONTROL operation 3 to rewind and unload the tape and calls SETMODE to disable function 120.

```
LITERAL WRITE^EOF = 2,
        REWIND^AND^UNLOAD = 3,
        RETURN^ERROR^IF^EOT = 120,
        ON = 1,
        OFF = 0;
 .
 .

CALL SETMODE(TAPE^NUM,
             RETURN^ERROR^IF^EOT,
             ON);
IF <> THEN ...
WHILE NOT DONE
BEGIN
    .
    .
   CALL WRITEX(TAPE^NUM,
               SBUFFER,
                  WCOUNT);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TAPE^NUM,
                            ERROR);
         IF ERROR = 150
         BEGIN
            CALL CONTROL(TAPE^NUM,
                         WRITE^EOF);
            IF <> THEN
            BEGIN
               CALL FILE_GETINFO_(TAPE^NUM,
                                  ERROR);
               IF ERROR <> 150 THEN ...
            END;

            CALL CONTROL(TAPE^NUM,
                         WRITE^EOF);
            IF <> THEN
            BEGIN
               CALL FILE_GETINFO_(TAPE^NUM,
                                  ERROR);
               IF ERROR <> 150 THEN ...
            END;

            CALL CONTROL(TAPE^NUM,REWIND^AND^UNLOAD);
            IF <> THEN ...
            CALL SETMODE(TAPE^NUM,
                         RETURN^ERROR^IF^EOT,
                         OFF);
            IF <> THEN ...
         END;
       .
       .
      IF <no more to write> THEN DONE = 1;
   END;
```

Reading From a Single-File Unlabeled Tape

When reading records from a tape reel containing one file, your program must do the following:

1. Space forward or backward to the record that you intend to read. If you try to space backward too far, the tape stops at the BOT sticker and the file system returns error 154. If the program is allowed to continue, it will then read the first record. If you try to space forward too far, the CONTROL procedure will return an end-of-file error (error 1). For example:

```
LITERAL SPACE^FWD^RECORDS = 9;
 .
 .
NUMBER^OF^RECORDS := 27;
CALL CONTROL(TAPE^FILE,
             SPACE^FWD^RECORDS,
             NUMBER^OF^RECORDS);
IF <> THEN
BEGIN
   CALL FILE_GETINFO_(TAPE^FILE,
                      ERROR);

   CASE ERROR OF
   BEGIN
      "1" -> ...                                   !end of file
      "154" -> ...                                 !BOT
      OTHERWISE -> ...                             !other error
   END;
END;
```

2. Read the record:

```
CALL READX(TAPE^FILE,
           SBUFFER,
           RCOUNT,
           COUNT^READ);
IF <> THEN ...
```

Reading From a Multiple-File Unlabeled Tape

Reading records from a multiple-file unlabeled tape reel is similar to the single-file case except that your program must also space the tape forward or backward to the appropriate file. The steps your program must perform are outlined below:

1. File space forward or backward to the appropriate file. If you try to space backward too far, the tape will stop when you reach the BOT sticker. To guard against spacing forward too far, however, your program should check for the pair of EOF marks that terminate information on the tape. For example:

   ```
   LITERAL SPACE^FWD^FILES = 7,
           SPACE^FWD^RECORDS = 9;
    .
    .
   NUMBER^OF^FILES := 5;
   CALL CONTROL(TAPE^NUM,
                SPACE^FWD^FILES,
                SPACE^FWD^RECORDS);
   IF <> THEN ...

   CALL READX(TAPE^NUM,
              SBUFFER,
              RCOUNT,COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPE^NUM,
                         ERROR);
      IF ERROR = 1 THEN ...             !end of tape reached
      ELSE ...                          !other error
   END;
   ```

2. Record space forward to the record that you intend to read. If you try to space forward too far, the CONTROL procedure returns an end-of-file error (error 1). For example:

   ```
   NUMBER^OF^RECORDS := 27;
   CALL CONTROL(TAPE^FILE,
                SPACE^FWD^RECORDS,
                NUMBER^OF^RECORDS);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPE^FILE,
                         ERROR);
      IF ERROR = 1 THEN ...                   !end of file
      ELSE ...                                !other error
   END;
   ```

3. Read the record:

   ```
   CALL READX(TAPE^FILE,
              SBUFFER,RCOUNT,
              COUNT^READ);
   IF <> THEN ...
   ```

Reading From a File on Multiple Unlabeled Tape Reels

The technique for reading records from a multiple-reel file depends in part on what information you have put in the tape header. Typically, your program is going to use the header information to determine whether the current reel is the first or last reel in the file. The program needs to know this to determine how to interpret the pair of EOF marks at the end of the information on the tape or how to interpret the BOT sticker.

If your program is record spacing forward and encounters the two EOF marks, the program should return a “record not found” message to the user if the current tape reel is the last in the file. Otherwise, the program should issue CONTROL function 24 to rewind the tape and request the next tape. Similarly, if the program encounters the BOT sticker, it needs to know whether to request that the user mount the previous tape.

Terminating Tape Access

You terminate access to a tape drive either by closing the device or by stopping the application. The technique is the same for labeled and unlabeled tapes: you close a tape device as you would any file, by issuing a FILE_CLOSE_ procedure call.

We recommend rewinding and unloading the tape before closing the file. You can do this by specifying 0 (the default value) for the *tape-disposition* parameter of FILE_CLOSE_. (You can also do it through the CONTROL procedure). You should turn off buffered mode before closing the tape device to ensure that all data is written to tape.

The following code fragment turns off buffered mode, rewinds and unloads the tape, and closes the tape file:

```
LITERAL BUFFERED^MODE = 99,
        ON            =  1,
        OFF           =  0,
        REWIND^UNLOAD =  0;
 .
 .
CALL SETMODE(TAPE^NUM,
             BUFFERED^MODE,
             OFF);
IF <> THEN ...

ERROR := FILE_CLOSE_(TAPE^NUM,
                     REWIND^UNLOAD);
IF ERROR <> 0 THEN ...
```

Note. With unlabeled tape, the application does not wait for the rewind-and-unload operation to complete; with labeled tape, the application waits for the operation to complete.

Recovering From Errors

The tape process attempts automatic recovery for all tape I/O operations. However, it is the application's responsibility to ensure that the tape gets positioned correctly following an error. For example, if a power failure or other hardware error occurs while a tape read or write is taking place, it is indeterminate where the tape is positioned at the point of failure. If, for example, an error is returned from a write request, you may not know whether the write to tape started.

If an error is reported when operating in buffered mode the application cannot determine which I/O operation caused the error. During a sequence of buffered writes, for example, an error reported to the application by the tape process does not indicate which of the previous write requests failed. Therefore, to recover from the error, the application must reposition to the last known good record and resume writing from that point.

In summary, your application must be able to respond to I/O errors depending on mode of buffering as follows:

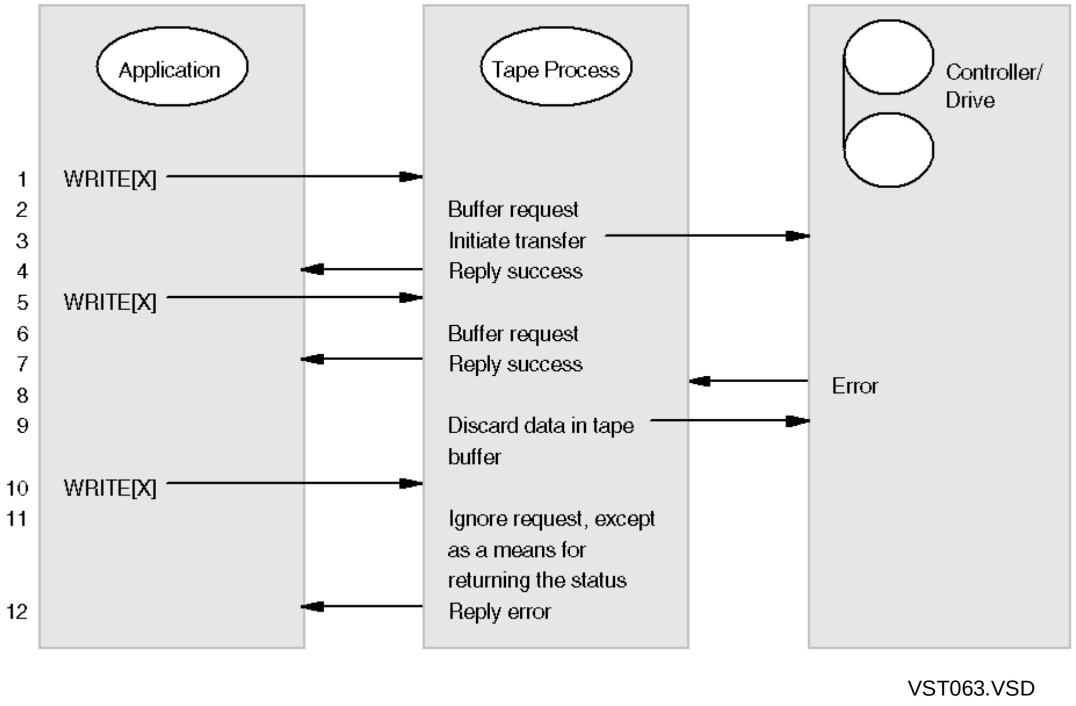
If you are using buffered mode...	Then your program must...
0 (no buffered mode)	Retry the current record
1 (buffered mode, no EOF mark buffering)	Be able to reconstruct the file on tape
2 (EOF mark buffering enabled)	Be able to reconstruct all the data it has written to tape

Reconsider the buffered-mode tape access example shown in Figure 12-3. Assume that the application encounters an error because the tape drive is unloaded and offline. Figure 12-4 shows what happens.

As described earlier in Working in Buffered Mode, two write requests are buffered by the tape process while the first write request is passed to the tape device. This time, however, the tape device replies with an error (error 100, for example). On receipt of an error, the tape process discards the contents of its buffer. The application is not notified of the error until the next time it passes a request to the tape process. In this case, the third write request receives the error. A CONTROL or SETMODE request to the tape process would also receive the error.

The application cannot tell from the information returned which write request caused the error.

Figure 12-4. Example of a WRITE Error in Buffered Mode


Application
Tape Process
Controller/ Drive
1 WRITE[X]
2 Buffer request
3 Initiate transfer
4 Reply success
5 WRITE[X]
6 Buffer request
7 Reply success
8 Error
9 Discard data in tape buffer
10 WRITE[X]
11 Ignore request, except as a means for returning the status
12 Reply error
VST063.VSD

The most commonly encountered tape errors are listed in Table 12-4. See the *Guardian Procedure Errors and Messages Manual* for a complete list of all file-system errors.

The *Guardian Procedure Errors and Messages Manual* provides details on the cause and effect of each of these errors, as well as the recommended action. For many of these errors, the required action is to simply print a message or repeat the operation. However, the following problems require special attention:

- “Device not ready” errors
- Power failure to the tape drive
- Path errors

Recovery from these problems is discussed in the following paragraphs.

Table 12-4. Commonly Encountered Tape Errors

File System Error Number	Caused by...
1	Read reached end of file (EOF) or write reached end of tape (EOT)
2	Invalid operation for a tape drive
21	Illegal count specified; attempt was made to transfer large data or not enough data
28	Too many outstanding nowait operations
40	Operation timed out
60	Device down or not open
66	Device down
100	Device not ready or controller not operational
101	Tape write protection is on
120	Data parity error, or attempt to access a tape whose density is higher than the switch setting on the tape drive
150	End of tape sticker detected
151	Runaway tape condition or incorrect tape density
153	Tape drive power on
154	BOT encountered during backspace files or backspace records
156	Tape command rejected
188	Data lost, especially when buffering is on
190	Device error; hardware problem
193	Invalid or missing microcode file
214	Channel timeout (hardware error)
218	Interrupt timeout
224	Controller error
200 through 255	Path errors

Recovering From “Device Not Ready” Errors

The system returns error 100 for several reasons: the tape unit is not powered up, there is no tape on the drive, or the tape is currently rewinding. Your program should retry the operation when the errant condition is fixed. Typically the fix requires human intervention, so it would be appropriate to prompt the user before retrying the operation.

Recovering From Tape Unit Power Failure

If the power to a magnetic tape unit fails and the application attempts a read, write, or control operation, then one of the following errors is returned: file-system error 100 (device not ready) or error 218 (interrupt timeout). After power is restored and the tape unit is again accessed, a subsequent call to FILE_GETINFO_ returns error 153 (tape drive power on). It is the responsibility of the application to ensure the correct tape is loaded following a power failure.

Tape units, if a tape is loaded, are automatically put back into an operating (ready) state when power is restored.

The position of the tape following power restoration depends on the drive type. You must therefore use care if you need to ensure that your code is device independent. Vacuum drives, for example, will move the tape when the power is lost; these drives automatically rewind the tape when power is restored. Some types of drives do not move the tape when power is lost; for these drives you can continue without having to reposition the tape.

Recovering From Path Errors

The system software usually corrects for path errors by finding an alternate path to the device, unless it is the tape unit itself that has the problem.

Typically, your program will retry the operation. Exactly what else the program must do, however, depends on whether the tape may have moved and on whether your program is executing in buffered mode.

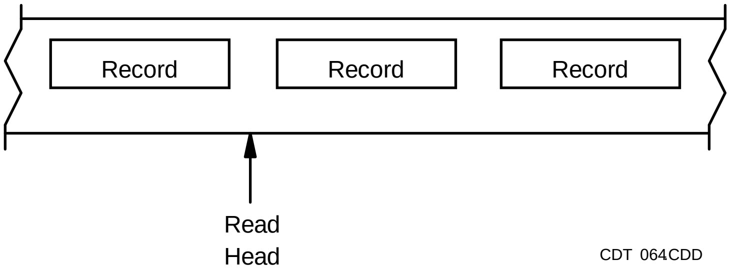
If error 200 or 201 is detected, the operation never got started. You can simply retry the operation for an application that is not executing in buffered mode. If the application is executing in buffered mode, there is no way to tell which operation initiated the error. Your program must backtrack to a known point of consistency and then play back all operations done since that time. Refer to Working in Buffered Mode, earlier in this section.

If your application detects an error in the range 210 through 231, then the operation failed at some indeterminate point. Tape motion may have occurred. These failures cause the tape process to switch to its backup process, so the next operation is tried in the alternate CPU.

There are several ways to handle path errors in the range 210 through 231:

- If a path error occurs during a write, space backward one record and read the record. If a parity error occurs, then it is clear that the write had been partially completed; backspace again and retry the write operation. If there is no parity error, then the write finished successfully.
- Keep track of the number of records read or written. Then, if an error of this type occurs, rewind the tape and space forward the appropriate number of records and reinitiate the operation.
- If writing, write a sequence number as part of each record written. If one of these errors occurs, retry the operation and continue. Then when reading the tape, discard all but the last record containing duplicate sequence numbers.

2 If you are reading and sequence numbers were written on tape, keep track of the sequence number of the current record. Then if a path error occurs, retry the operation. If the expected sequence number is not read, meaning that a record was skipped over when the path error occurred, backspace the tape *x* records, where *x* is the sequence number on tape minus the last known sequence number. For example, assume the current sequence number is 3 and you issue another read request:



The read request returns a path error. The tape may have moved forward one record or it may have stayed where it was. Now retry the read request. If the request returns the record with sequence number 4, then the tape did not move and you have the record you wanted. If the read returns the record with sequence number 5, then the tape did move. You now need to space the tape backward two records (5 minus 3) to read the record with sequence number 4.

Accessing an Unlabeled Tape File: An Example

This subsection shows a sample program that performs an application similar to the labeled-tape program earlier in this section. The difference is that this program uses unlabeled tape. Major coding differences are as follows:

- The program opens the magnetic tape device directly by name.
- The program must do its own file positioning. That is, the tape does not get automatically positioned at the beginning or end of a file when opened. The program must use CONTROL operations to position the tape.
- When writing to a scratch tape, the program can initialize the tape itself by simply writing two EOF marks after the BOT sticker. A separate function selects procedure SCRATCH^TAPE to do this.
- The program performs additional error checking following errors that might occur during reading or writing the tape file. A sequence number included in each record makes this possible. The following procedures provide this error checking:
    - The TAPE^WRITE^ERRORS procedure is called by APPEND^RECORD whenever an error occurs on writing to tape. This procedure displays a message telling the user that an error has occurred and also prints the file-system error number. Because the program uses buffered mode, however, the

program does not know which write operation caused the error. Therefore this procedure backspaces the tape to the last correctly written record and displays it. The user then has the option of reentering the data submitted since the displayed record or exiting the program.

- The TAPE^READ^ERRORS procedure is called by READ^RECORD if an error is encountered when a record block is read from tape. Here, the program does not know whether the tape moved, so TAPE^READ^ERRORS reads the next record block and examines the sequence number put on the tape when the record was written. If the sequence number is one greater than the current sequence number, then the tape did not move and the record just read is the one the user wants. If the number is two greater, then the tape had moved; the procedure discards the record just read, winds the tape back two records, and reads again.

The code for the program follows.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

LITERAL  BUFSIZE  = 512;
LITERAL  TBUFSIZE = 2048;
LITERAL  MAXFLEN  = ZSYS^VAL^LEN^FILENAME;
LITERAL  ABEND    = 1;

STRING  .SBUFFER[0:BUFSIZE];        !Buffer for terminal I/O
INT      TERMNUM;                   !Terminal file number
STRING  .S^PTR;

INT     .TBUFFER[0:(TBUFSIZE/2) - 1];   !Buffer for tape I/O
INT     .LREC0 := @TBUFFER[0];      !Integer pointers to
INT     .LREC1 := @TBUFFER[256];    ! records in tape
INT     .LREC2 := @TBUFFER[512];    ! buffer
INT     .LREC3 := @TBUFFER[768];

INT      INDEX;                     !Index into record block
INT(32)  RBLOCK;                    !Record block number
INT      SEQ^NUM;                   !Record block sequence
                                    ! number
INT      TAPENUM;                   !Tape file number

STRUCT  .LOG^RECORD;                !Record structure
BEGIN
   STRING   DATE[0:7];
   INT      SEQ^NUM;
   STRING   COMMENTS[0:501];
END;
```

```
INT     .RECORD^POINTER := @LOG^RECORD[0];

STRUCT  .CI^STARTUP;                !Startup message
BEGIN
   INT   MSGCODE;
   STRUCT   DEFAULT;
   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
   END;
   STRUCT   INFILE;
   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
      INT   FILEID[0:3];
   END;
   STRUCT   OUTFILE;

   BEGIN
      INT   VOLUME[0:3];
      INT   SUBVOL[0:3];
      INT   FILEID[0:3];
   END;
   STRING   PARAM[0:529];
END;

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,
?   WRITEREADX,WRITEX,PROCESS_STOP_,READX,CONTROL,DNUMOUT,
?   FILE_GETINFO_,DNUMIN,SETMODE,OLDFILENAME_TO_FILENAME_)
?LIST
```

```
!------------------------------------------------------------
! Here are some DEFINEs to make it easier to format and print
! messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N) =
          @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE PRINT^LINE =
          CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE PRINT^BLANK =
          CALL WRITE^LINE(SBUFFER,0)   #;

!  Print a string:

   DEFINE PRINT^STR(S) = BEGIN START^LINE;
                               PUT^STR(S);
                               PRINT^LINE;  END  #;
```

```
!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there is no file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING   .FNAME;
INT       LEN;
INT       ERROR;

BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program

   CALL PROCESS_STOP_(!process^handle!,
                      !specifier!,
                      ABEND);
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and the error number are determined from the file
! number and FILE^ERRORS^NAME is then called to do the
! display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT          FNUM;

BEGIN
   INT       ERROR;
   STRING   .FNAME[0:MAXFLEN - 1];
   INT       FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!------------------------------------------------------------
! Procedure to write a message on the terminal and check for
! any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!-------------------------------------------------------------
! Procedure for responding to errors incurred while reading
! from magnetic tape. This procedure tries the read again. If
! sequence numbers are inconsistent, then a read was skipped
! due to the error.  The procedure compensates by backspacing
! over two records.
!-------------------------------------------------------------

PROC TAPE^READ^ERRORS(ERR^NO);
INT      ERR^NO;

BEGIN
   INT   COUNT^READ;

!  Set up the buffer and display error number on terminal:

   PUT^STR
      ("Tape Read Error: File System Error Number Is:   ");
   PUT^INT(ERR^NO);
   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Reissue the read call:

   CALL READX(TAPENUM,TBUFFER,TBUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Extract a record:

   LOG^RECORD[0] ':=' TBUFFER[0] FOR 256;

!  Check the sequence number. If it is one greater than
!  the current sequence number, then the program has read the
!  intended record. If it is two greater, then the program
!  skipped a record block on account of the error. In this
!  case, the procedure backspaces two records and then reads
!  again. If the sequence number is neither one nor two
!  greater, then the program cannot establish the correct
!  record by this means (this will happen, for example, if
!  an error occurs during the first read after positioning
!  the tape).

   IF LOG^RECORD.SEQ^NUM = (SEQ^NUM +1) THEN
   BEGIN
   END
   ELSE IF LOG^RECORD.SEQ^NUM = (SEQ^NUM + 2)
   THEN
      !Do nothing
   BEGIN
      CALL CONTROL(TAPENUM,10,2);
      IF <> THEN CALL FILE^ERRORS(TAPENUM);
      CALL READX(TAPENUM,TBUFFER,TBUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TAPENUM);
   END
   ELSE
   BEGIN
      PRINT^STR("Read error: Unable to Verify sequence.");
      SBUFFER ':=' "Do You Wish to Continue? (y/n)  "
                   -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                     BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      IF NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y") THEN
      CALL PROCESS_STOP_;
   END;
END;


!-------------------------------------------------------------
! Procedure for responding to errors incurred when writing to
! magnetic tape.  Because the program uses buffered mode, the
! program does not know which records have been written to
! tape. This procedure therefore backspaces the tape to the
! last record block that was correctly written to tape,
! then prompts the user to reenter missing messages.
!-------------------------------------------------------------

PROC TAPE^WRITE^ERRORS(ERR^NO);
INT       ERR^NO;

BEGIN
   INT    COUNT^READ;

!  Set up the buffer and display error number on terminal:

   START^LINE;
   PUT^STR
      ("Tape Write Error: File System Error Number is:  ");
   PUT^INT (ERR^NO);
   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Space the tape backward one record:

   CALL CONTROL(TAPENUM,10,1);

!  Read the record. If the read returns an error, space back
!  to the previous record and read that one. If that read
!  returns an error, call FILE^ERRORS and stop the program:

   CALL READX(TAPENUM,TBUFFER,TBUFSIZE);
   IF <> THEN
   BEGIN
      CALL CONTROL(TAPENUM,10,2);
      CALL READX(TAPENUM,TBUFFER,TBUFSIZE);
      IF <> THEN  CALL FILE^ERRORS(TAPENUM);

   !  Get the last record in the retrieved record block:

      LOG^RECORD[0] ':=' LREC3[0] FOR 256;
```

```
    !  Display date from last good record:

       CALL WRITEX(TERMNUM,LOG^RECORD.DATE,
                   $LEN(LOG^RECORD.DATE));
       IF <> THEN CALL FILE^ERRORS(TERMNUM);

    !  Display comments from last good record:

       CALL WRITEX(TERMNUM,LOG^RECORD.COMMENTS,
                   $LEN(LOG^RECORD.COMMENTS));
       IF <> THEN CALL FILE^ERRORS(TERMNUM);

    !  Prompt user to continue:

       SBUFFER ':=' "Do You Wish to Continue(y/n)  "
                    -> @S^PTR;
       CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                       BUFSIZE,COUNT^READ);
       IF <> THEN CALL FILE^ERRORS(TERMNUM);

    !  If the user indicates continue, prompt to reenter data,
    !  then return to APPEND^RECORD procedure:

       IF SBUFFER[0] = "y"
       THEN
       BEGIN
          SBUFFER ':=' "Please Reenter Your Data." -> @S^PTR;
          CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
          IF <> THEN CALL FILE^ERRORS(TERMNUM);

       !  Reset the sequence number so that subsequent
       !  records are correctly sequenced:

          SEQ^NUM := LOG^RECORD.SEQ^NUM;
       END

    !  If user declines to continue, stop the program:

       ELSE
       CALL PROCESS_STOP_;
    END;
 END;
```

```
!------------------------------------------------------------
! Procedure to rewind the tape to BOT, checking that the
! tape is loaded.  If not, then the rewind operation
! results in error 100.  The user is prompted to load the
! tape before continuing.
!------------------------------------------------------------

PROC LOAD^TAPE;
BEGIN
   INT     ERROR;
   INT     COUNT^READ;

CHECK^AGAIN:
   CALL CONTROL(TAPENUM,6);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPENUM,ERROR);
      IF ERROR = 100 THEN
      BEGIN
         SBUFFER ':=' ["Tape Not Ready.  Press RETURN ",
                       "When Ready to Continue: "] -> @S^PTR;
         CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                         BUFSIZE,COUNT^READ);
         GOTO CHECK^AGAIN;
      END
      ELSE CALL FILE^ERRORS(TERMNUM);
   END;
END;
```

```
!------------------------------------------------------------
! This procedure executes when you press "r" in response to
! the function prompt in the main procedure. It prompts the
! user for the desired record, displays it on the terminal,
! then prompts for sequential reads.
!------------------------------------------------------------

PROC READ^RECORD;
BEGIN
   INT          COUNT^READ;
   INT(32)      RECORD^NUM;
   STRING .EXT  NEXT^ADDR;
   INT          STATUS;
   INT          ERROR;

!  Prompt the user to select a record:

PROMPT^AGAIN:
   PRINT^BLANK;
   SBUFFER ':=' "Enter Record Number:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
   SBUFFER[COUNT^READ] := 0;

!  Convert ASCII to numeric:

   @NEXT^ADDR := DNUMIN(SBUFFER,RECORD^NUM,10,STATUS);
   IF STATUS OR @NEXT^ADDR <> $XADR(SBUFFER[COUNT^READ])
   THEN
   BEGIN
      PRINT^STR("Error in the record number");
      GOTO PROMPT^AGAIN;
   END;

!  Calculate record block number, assuming blocking
!  factor of 4:

   RBLOCK := RECORD^NUM / 4D;

!  Modulo divide to get record index:

   INDEX := RECORD^NUM '\' 4;
```

```
!  Rewind tape to BOT, leave online. Since this might be the
!  first access to tape, the code retries the operation for
!  error 100.  The call to FILE^ERRORS prompts the user to
!  fix the problem before the retry:

   CALL LOAD^TAPE;

!  Space tape forward to start of record block:

   CALL CONTROL(TAPENUM,9,$INT(RBLOCK));
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

!  Execute loop if reading just selected, or user
!  has requested to read an additional record
!  Exit loop if user declines to read next record:

   DO BEGIN

      PRINT^BLANK;

   !  Read a record block from the tape file:

      CALL READX(TAPENUM,TBUFFER,TBUFSIZE,COUNT^READ);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TAPENUM,ERROR);
         IF ERROR = 1 THEN
            PRINT^STR("No such record. ")
         ELSE CALL TAPE^READ^ERRORS(ERROR);
         RETURN;
      END;

   !  Extract sequence number for record block from first
   !  record to help fix errors:

      LOG^RECORD ':=' TBUFFER[0] FOR 256;
      SEQ^NUM := LOG^RECORD.SEQ^NUM;

      DO BEGIN

      !   Extract the record:

         CASE INDEX OF
         BEGIN
            0 -> LOG^RECORD[0] ':=' LREC0[0] FOR 256;
            1 -> LOG^RECORD[0] ':=' LREC1[0] FOR 256;
            2 -> LOG^RECORD[0] ':=' LREC2[0] FOR 256;
            3 -> LOG^RECORD[0] ':=' LREC3[0] FOR 256;
            OTHERWISE -> CALL PROCESS_STOP_(!process^handle!,
                                            !specifier!,
                                            ABEND);
         END;
      !  Check for incomplete record block.  If this record
      !  is blank, set INDEX 4 in preparation for reading
      !  the next record block.  Also set SBUFFER to "Y" in
      !  case this is the first record selected:

         IF LOG^RECORD.SEQ^NUM = "  " THEN
         BEGIN
            INDEX := 4;
            SBUFFER[0] ':=' "Y";
         END
         ELSE

      !  Process the log record:

         BEGIN

         !  Display date from the record:

            CALL WRITEX(TERMNUM,LOG^RECORD.DATE,8);
            IF <> THEN CALL FILE^ERRORS(TERMNUM);

         !  Display comments and increment the record
         !  counter:

            CALL WRITEX(TERMNUM,LOG^RECORD.COMMENTS,502);
            IF <> THEN CALL FILE^ERRORS(TERMNUM);
            INDEX := INDEX + 1;

         !  Prompt the user to read the next record:

            PRINT^BLANK;
            SBUFFER ':='
               "Do You Want To Read the Next Record (y/n)  "
               -> @S^PTR;
            CALL WRITEREADX(TERMNUM,SBUFFER,
                            @S^PTR '-' @SBUFFER,
                            BUFSIZE,COUNT^READ);
            IF <> THEN CALL FILE^ERRORS(ERROR);
         END;
      END
      UNTIL (NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y"))
             OR INDEX = 4;

   !  No more records in this record block.  Reset record
   !  count to 0 and read next record block, if requested:

      INDEX := 0;
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
END;
```

```
!------------------------------------------------------------
! Procedure to append a record to the file.
!------------------------------------------------------------

PROC APPEND^RECORD;
BEGIN
   INT   ERROR;
   INT COUNT^READ;

!  Rewind tape and position the tape to EOF. Since this might
!  be the first access to tape, check whether the tape is
!  ready by checking for error 100. If error 100 detected,
!  call FILE^ERRORS to prompt the user to fix the problem,
!  then retry the operation:

   CALL LOAD^TAPE;

!  Position the tape to EOF:

   CALL CONTROL(TAPENUM,7,1);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

   CALL CONTROL(TAPENUM,8,1);
   IF <> THEN  CALL FILE^ERRORS(TAPENUM);

!  Space back one record and establish sequence number of
!  last record written to tape:

   CALL CONTROL(TAPENUM,10,1);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(TAPENUM,ERROR);
      IF ERROR = 154
      THEN SEQ^NUM := 0
      ELSE CALL FILE^ERRORS(TAPENUM);
   END
   ELSE
   BEGIN
      CALL READX(TAPENUM,TBUFFER,TBUFSIZE);
      IF <> THEN CALL FILE^ERRORS(TAPENUM);

      LOG^RECORD ':=' TBUFFER[0] FOR 512;
      SEQ^NUM := LOG^RECORD.SEQ^NUM + 1;
   END;

!  Blank tape buffer:

   TBUFFER[0] ':=' "  ";
   TBUFFER[1] ':=' TBUFFER[0] FOR 1023;

!  Initialize the index into the tape buffer:

   INDEX := 0;
```

```
!  Write records to file.  This loop prompts the user for
!  each additional record you want to write:

   DO BEGIN

   !  Blank the log record structure:

      RECORD^POINTER[0] ':=' "  ";
      RECORD^POINTER[1] ':=' RECORD^POINTER[0] FOR 255;

   !  Prompt user for date:

   PROMPT^AGAIN:
      PRINT^BLANK;
      SBUFFER ':=' "Enter Today's Date (mmddyyyy):  "
                   -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      IF COUNT^READ <> 8 THEN GOTO PROMPT^AGAIN;

   !  Put date into record structure:

      LOG^RECORD.DATE ':=' SBUFFER[0] FOR (COUNT^READ);

   !  Prompt user for comments:

      SBUFFER ':=' "Please Enter Your Comments: " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

   !  Put comments into record structure:

      LOG^RECORD.COMMENTS ':=' SBUFFER[0] FOR COUNT^READ;

   !  Put sequence number in record structure:

      LOG^RECORD.SEQ^NUM := SEQ^NUM;

   !  Pack record into record block:

      CASE INDEX OF
      BEGIN
         0 -> LREC0 ':=' LOG^RECORD FOR 256;
         1 -> LREC1 ':=' LOG^RECORD FOR 256;
         2 -> LREC2 ':=' LOG^RECORD FOR 256;
         3 -> LREC3 ':=' LOG^RECORD FOR 256;
         OTHERWISE -> CALL PROCESS_STOP_;
      END;
```

```
!  Prompt the user to enter additional records:

   PRINT^BLANK;
   SBUFFER ':='
           "Do You Wish to Enter Another Record (y/n)? "
           -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Increment the index into the record block:

   INDEX := INDEX + 1;

!  Send record block to tape process if no more
!  records, or if record block full. Flush out to tape
!  every 10 writes to provide known point of consistency:

   IF INDEX = 4 OR
      (NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y")) THEN
   BEGIN
      CALL WRITEX(TAPENUM,TBUFFER,TBUFSIZE);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(TAPENUM,ERROR);
         IF ERROR <> 1 THEN CALL TAPE^WRITE^ERRORS(ERROR);
      END;

   !  Increment record block sequence number and reset
   !  the index:

      SEQ^NUM := SEQ^NUM + 1;
      INDEX := 0;

   !  Blank tape buffer in case next record block is not
   !  full:

      TBUFFER[0] ':=' "  ";
      TBUFFER[1] ':=' TBUFFER[0] FOR 1023;

   !  Flush to tape every 10 record blocks. Use modulo
   !  divide to detect tenth record. Buffered mode is
   !  already set, therefore SETMODE 99 forces to tape all
   !  records in tape buffer:

      IF $DBL(SEQ^NUM) '\' 10 = 0 THEN
      BEGIN
         CALL SETMODE(TAPENUM,99,1);
         IF <> THEN CALL FILE^ERRORS(TAPENUM);
      END;
   END;
END
UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");
```

```
!  Write EOF marks to end of tape:

   CALL CONTROL(TAPENUM,2);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
   CALL CONTROL(TAPENUM,2);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
END;

!------------------------------------------------------------
! Procedure to initialize a scratch tape with two EOF marks.
!------------------------------------------------------------

PROC SCRATCH^TAPE;

BEGIN

!  Make sure tape is at BOT. Because this may be the first
!  access to tape, check for error 100. If detected, call
!  FILE^ERRORS to prompt user to fix the problem, then retry:

   CALL LOAD^TAPE;

!  Write two EOF marks to tape:

   CALL CONTROL(TAPENUM,2);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

   CALL CONTROL(TAPENUM,2);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

!  Rewind to BOT ready for writing records.

   CALL CONTROL(TAPENUM,6);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
END;

!------------------------------------------------------------
! Procedure to stop the program on request.  As well as
! stopping the program, this procedure rewinds and unloads
! the tape.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN

!  Rewind and unload the tape:

   CALL CONTROL(TAPENUM,3);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);

!  Stop the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure to process an illegal command.  The procedure
! informs the user that the selection was other than "r,"
! "a," "i," or "x."
!------------------------------------------------------------

PROC ILLEGAL^COMMAND;

BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid then
!  return to prompt again for a valid function:

   PRINT^STR("ILLEGAL COMMAND: " &
             "Type one of 'r,' 'a,' 'i,' or 'x.'");
END;


!------------------------------------------------------------
! Procedure to prompt the user for the next function to be
! performed:
!
!  "r" to read records
!  "a" to append records
!  "i" to initialize a scratch tape
!  "x" to exit the program
!
! The selection made is returned as the result of the call.
!------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("     'a' for Append to Log, ");
   PRINT^STR("     'i' for Initialize a Scratch Tape, ");
   PRINT^STR("     'x' for Exit. ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to save Startup message in a global structure.
!------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH, MATCH) VARIABLE;

INT    .RUCB;
INT    .START^DATA;
INT    .MESSAGE;
INT     LENGTH;
INT     MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;
```

```
!-------------------------------------------------------------
! Procedure to perform initialization for the program.  It
! calls INITIALIZER to read and copy the Startup message
! into the global variables s area and then opens the IN file
! specified in the Startup message.  This procedure also
! opens the tape file and sets buffered mode for tape access.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .TAPE^NAME[0:MAXFLEN - 1];
   STRING .TERM^NAME[0:MAXFLEN - 1];
   INT     TAPELEN;
   INT     TERMLEN;
   INT     OPEN^FLAG;
   INT     ERROR;

!  Read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
            CI^STARTUP.INFILE.VOLUME,
            TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                        !specifier!,
                                        ABEND);
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                        !specifier!,
                                        ABEND);

!  Open the tape file for exclusive access:

   ERROR := OLDFILENAME_TO_FILENAME_(
            CI^STARTUP.OUTFILE.VOLUME,
            TAPE^NAME:MAXFLEN,TAPELEN);
   IF ERROR <> 0
      THEN CALL FILE^ERRORS^NAME(TAPE^NAME:TAPENUM,ERROR);
   ERROR := FILE_OPEN_(TAPE^NAME:TAPELEN,TAPENUM,
                       !access!,
                       1);
   IF ERROR <> 0
      THEN CALL FILE^ERRORS^NAME(TAPE^NAME:TAPELEN,ERROR);

!  Set buffered mode:

   CALL SETMODE(TAPENUM,99,1);
   IF <> THEN CALL FILE^ERRORS(TAPENUM);
END;
```

```
!------------------------------------------------------------
! This is the main procedure
!------------------------------------------------------------

PROC LOGGER MAIN;
BEGIN
   STRING CMD;

   CALL INIT;

!  Loop indefinitely until user selects function x:

   WHILE 1 DO
   BEGIN

   !  Prompts for the next function to perform:

      CMD := GET^COMMAND;

   !  Call function selected by user:

      CASE CMD OF
      BEGIN

         "r", "R" -> CALL READ^RECORD;

         "a", "A" -> CALL APPEND^RECORD;

         "i", "I" -> CALL SCRATCH^TAPE;

         "x", "X" -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL ILLEGAL^COMMAND;
      END;
   END;
END;
```

13 Manipulating File Names

This section describes how an application program can manipulate file names or the names of entities, such as nodes or volumes, that make up parts of file names. See Section 2, Using the File System, if you are unsure about the format of file names.

A typical use of the features described here is to manipulate file names or file-name patterns presented to a program using a TACL command. You can use these features to check or process the file names or patterns received in the Startup or Param message. A program listing at the end of this section shows an example.

This section discusses how to perform the following operations on file names:

- Scan a string of characters to find out whether it contains a valid file name (FILENAME_SCAN_ procedure).
- Resolve a file name into its fully qualified form (FILENAME_RESOLVE_ procedure).
- Reduce a file name to its shortest usable form by removing the default node name, volume, or subvolume portions (FILENAME_UNRESOLVE_ procedure).
- Extract selected portions of a file name (FILENAME_DECOMPOSE_ procedure).
- Modify portions of a file name (FILENAME_EDIT_ procedure).
- Compare two file names to see whether they identify the same object (FILENAME_COMPARE_ procedure).
- Search for file names using file-name patterns (FILENAME_FINDSTART_, FILENAME_FINDNEXT_, FILENAME_FINDFINISH_, and FILENAME_MATCH_ procedures).

Overview

The procedures described in this section can manipulate file names for disk files and device files alike. Process file names can also be manipulated by the procedures listed above.

Many of the procedures listed above can also be used with DEFINE names, as described later in this section.

Refer to Section 16, Creating and Managing Processes, for information about procedures that manipulate process handles.

Identifying Portions of File Names

Many of the procedures described in this section need to identify portions of file names or names of other entities such as nodes, volumes, and subvolumes. For example, if you want to change a subvolume name in a permanent disk-file name, you need a way of specifying to a procedure that the string you supply is to replace the subvolume name.

To describe how you identify portions of entity names to the procedures described here, the following paragraphs introduce some terminology.

Defining a File-Name “Part”

A file-name *`part`* represents a portion of a file name either between two periods, before the first period, or after the last period. Node name, file ID, process name, process qualifier, and device name are all examples of file-name parts.

To identify a part of a file name, many procedures take a *`level`* parameter. The level identifies the position of the part in the file name. The level can have a value between -1 and +2 as follows:

-1 Identifies a node name.

0 Identifies that part of a file name that immediately follows the node name in a fully qualified file name; for example, a device name, volume name, or unqualified portion of a process name. This part often has a dollar sign ($) as the first character.

1 Identifies the first qualifier. For a disk file, this is the subvolume name. For a process, this is the first process qualifier.

2 Identifies the second qualifier. For a disk file, this is the file ID. For a process file, this is the second process qualifier.

Some examples of file names are listed below. The examples indicate the level number of each part of the file name:

Fully qualified permanent disk-file name:	\SYSA.$OURVOL.MYSUB.PROGA
Level:	-1 0 2
Partially qualified disk file:	PROGA
Level:	2
Named process file name:	$SERV.#Q1
Level:	0 1
Unnamed process descriptor:	\SYSA.$:15:132:3
Level:	-1 0
Printer file:	$LP1
Level:	0

Defining a File-Name “Piece”

A *`piece`* of a file name contains one or more consecutive parts of a file name. Many file-name manipulation procedures require a *`piece`* parameter. When a file-name piece consists of just one part, the level parameter is enough to specify the desired part.

You can supply a file-name *`suffix`* as a piece. Here, the piece consists of the part identified by the *`level`* parameter plus all parts to the right of that part.

Similarly, you can specify a *`prefix`* as a file name piece. Here, the piece consists of the part identified by the *`level`* parameter and all parts to the left of that part.

To specify the suffix or prefix, you use the *`options`* parameter of the procedure. The following example shows the use of level numbers to identify file-name pieces:

File name:	\SYSA.$OURVOL.MYSUB.PROGA
Level 1 piece:	MYSUB
Level 1 piece with suffix:	MYSUB.PROGA
Level 1 piece with prefix:	\SYSA.$OURVOL.MYSUB

Defining a File-Name “Subpart”

Some file-name parts split into smaller elements called *`subpart`*s. This applies to named and unnamed process descriptors. A subpart is an element of a part separated from the next subpart by a colon (:). For example, the level 0 part of an unnamed process descriptor is made up of a dollar sign ($), a IPU designator, a process identification number (PIN), and a sequence number. These subparts are separated by colons.

Some procedures, such as FILENAME_DECOMPOSE_, accept a *`subpart`* parameter.

Working With File-Name Patterns

The procedures described in this section deal not only with file names, but also with file-name patterns that contain asterisk (*) and question mark (?) wild-card characters. These wild-card characters have the following meanings:

*	Matches zero or more letters, digits, dollar signs ($), or pound signs (#)
?	Matches exactly one letter, digit, dollar sign ($), or pound sign (#)

Wild-card characters can appear in any part of a name, as many times as there can be characters in the part. Because an asterisk can match zero characters, the pattern of a file-name part can be twice the size of the corresponding file-name part, including ordinary characters and wild cards. For example, a subvolume pattern could be 16 characters long.

The following examples show the use of wild-card characters:

Z	Matches all file names containing the letter Z in the current subvolume
$S.*	Matches all locations of the spooler on the current system
..*	Matches all permanent files on the current system, as well as processes with two qualifiers
*.$DATA	Matches $DATA on all systems
*	Matches all systems
Z?	Matches all two-character file names in the current volume that start with the letter Z

Scanning, Resolving, and Unresolving File Names

This subsection discusses the following operations on file names:

- How to use the FILENAME_SCAN_ procedure to test that a string contains a valid file name or file-name pattern.
- How to use the FILENAME_RESOLVE_ procedure to expand a file name or file-name pattern into a fully qualified file name using default values for the node, volume, and subvolume names.
- How to use the FILENAME_UNRESOLVE_ procedure to remove any part of the file name or file-name pattern that is currently part of the default values.
- How to use the FILENAME_DECOMPOSE_ procedure to extract elements of a file name or file-name pattern; for example, to extract the volume name from a file name.

Scanning a String for a Valid File Name

To check that a given character string contains a valid entity name (or name pattern), you use the FILENAME_SCAN_ procedure. You typically do this before using a file name in any other way. FILENAME_SCAN_ is the only procedure for manipulating file names that completely checks the validity of an input file name.

Scanning file names checks that the sizes of file-name parts or file-name-pattern parts are valid and that the tested substring is immediately followed by a character that is not allowed in a file name or file-name pattern. Note that only the syntactic correctness of the name is checked; no attempt is made to check for the existence of the named entity.

You pass a string to the FILENAME_SCAN_ procedure for testing. If the file name or pattern is valid, the procedure returns the following information:

- The length in bytes of the file name or pattern.
- An indication of the kind of object that the name or pattern identifies: file name, file-name pattern, or DEFINE name.
- The level of the specified object, that is, whether the name identifies a network node, device or process, subvolume, or disk-file name.

If the string does not contain a valid file name or file-name pattern, then the procedure returns error value 13 (illegal file name).

Note that the FILENAME_SCAN_ procedure does not check the entire input string. If the front part of the string contains a valid file name or pattern, then the rest of the string is ignored. If you need to check that the entire string has been tested, you should include a test that the string length is equal to the byte count returned by the FILENAME_SCAN_ procedure.

In addition to the default action of scanning for a file name, the FILENAME_SCAN_ procedure is also able to scan for a valid subvolume name or for a name pattern. The following paragraphs describe how.

Scanning File Names and Node Names

To scan a string for the existence of any kind of valid file name or node name, you use the FILENAME_SCAN_ procedure without any options. The default action of the FILENAME_SCAN_ procedure is to accept any syntactically valid file name (including disk-file name, DEFINE name, process file name, or device name) or node name; subvolume names are rejected, as are name patterns.

The following example scans a string in the first STRING^LENGTH bytes of STRING^BUFFER. The value of STRING^LENGTH is compared to the returned value of COUNT to verify that the entire string has been checked:

```
ERROR := FILENAME_SCAN_(STRING^BUFFER:STRING^LENGTH,
                        COUNT,
                        KIND,
                        LEVEL);
IF ERROR <> 0 OR
   COUNT <> STRING^LENGTH THEN ...       ! Error condition
```

Scanning File-Name Patterns

To accept a file-name pattern in the input string of the FILENAME_SCAN_ procedure, you need to set the `accept-pattern` flag (bit 15) in the `options` parameter to 1. The `options` parameter goes at the end of the parameter list:

```
LITERAL ACCEPT^PATTERNS = %B0000000000000001;
 .
 .
OPTIONS := ACCEPT^PATTERNS;
ERROR := FILENAME_SCAN_(STRING^BUFFER:STRING^LENGTH,
                        COUNT,KIND,
                        LEVEL,
                        OPTIONS);
IF ERROR <> 0 THEN ...                    ! Error condition
```

Scanning Subvolume Names

To accept subvolume names in the input string of the FILENAME_SCAN_ procedure, you need to set the `accept-subvol` flag (bit 14) in the `options` parameter to 1:

```
LITERAL ACCEPT^SUBVOLS = %B0000000000000010;
 .
 .
OPTIONS := ACCEPT^SUBVOLS;
ERROR := FILENAME_SCAN_(STRING^BUFFER:STRING^LENGTH,
                        COUNT,
                        KIND,
                        LEVEL,
                        OPTIONS);
IF ERROR <> 0 THEN ...                    ! Error condition
```

Scanning File Names: Some Examples

The following examples list valid input strings, assuming that your program chooses to accept file name patterns and subvolumes as valid input strings to the FILENAME_SCAN_ procedure:

String	**Count**	**Level**	**Comments**
\SYSA	6	-1	Valid form of node name
.$.*	6	1	Subvolume pattern
$VOLUME1.ACCOUNTS.OVERDUE,NAME	25	2	Last part of string ignored
$SERV	5	0	Process file name

Resolving Names

To resolve a name into its fully qualified form, you use the FILENAME_RESOLVE_ procedure. This procedure takes a string value containing a partially resolved name as its input, uses the default values to replace any missing parts, and then returns the fully qualified name.

In addition to the default action of resolving a file name, the FILENAME_RESOLVE_ procedure is able to resolve subvolume names; process and expand DEFINEs; override the current network node, volume, and subvolume default values; use a search DEFINE to find a name to resolve; or override the input file-name string with a DEFINE. The following paragraphs explain how.

△ **Caution.** Passing an invalid name or file-name pattern to the FILENAME_RESOLVE_ procedure can result in a signal, trap, or data corruption. To verify that a name is valid, use the FILENAME_SCAN_ procedure.

Resolving File Names

To resolve a partially qualified file name into its fully qualified form, you use the FILENAME_RESOLVE_ procedure without any options. The default action of this procedure is to expand any partially qualified file name provided in its input string. Subvolume names are not resolved without using special options. Moreover, no special processing of DEFINE names is done without the use of special options; DEFINE names are returned without change.

The following example shows typical use of the FILENAME_RESOLVE_ procedure in qualifying a file name checked by the FILENAME_SCAN_ procedure:

```
LITERAL MAXLEN = 256;
 .
 .
ERROR := FILENAME_SCAN_(STRING^BUFFER:STRING^LENGTH,
                        COUNT,KIND,LEVEL);
IF ERROR <> 0 THEN ...                    ! Error condition
ELSE
BEGIN
   ERROR := FILENAME_RESOLVE_(STRING^BUFFER:COUNT,
                              FULLNAME:MAXLEN,
                              FULL^LENGTH);
   IF ERROR <> 0 THEN ...                 ! Error condition
END;
```

Here, the FILENAME_RESOLVE_ procedure takes its string input in STRING^BUFFER and returns the fully qualified name in FULLNAME. The length of the fully qualified name is returned in FULL^LENGTH.

The following examples show how the FILENAME_RESOLVE_ procedure expands some file names. The examples assume default values of \SYSA.$OURVOL.MYSUB:

Input File Name	Output File Name
PROGA	\SYSA.$OURVOL.MYSUB.PROGA
$THEIRVOL.OLDSUB.PROGA	\SYSA.$THEIRVOL.OLDSUB.PROGA
\SYSB.$OURVOL.HERSUB.PROGA	\SYSB.$OURVOL.HERSUB.PROGA

Overriding the Default Values

The FILENAME_RESOLVE_ procedure usually obtains the default values for the node name, volume name, and subvolume name from the =_DEFAULTS DEFINE. However, you can override these values by specifying the *`defaults`* parameter. This parameter supplies the subvolume name itself and optionally the volume and node names. (If the volume or node name is omitted, then the corresponding values from the =_DEFAULTS DEFINE are used.) You can also use the *`defaults`* parameter to specify an alternate defaults class DEFINE.

The following example uses alternate defaults. The name string or defaults DEFINE would typically be a user-specified parameter to the program.

```
DEFAULTS ':=' PARAM2 FOR $LEN(PARAM2);
DEFAULTS^LENGTH := $LEN(PARAM2);

ERROR := FILENAME_RESOLVE_(PNAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           !options!,
                           !override^name:length!,
                           !search:length!,
                           DEFAULTS:DEFAULTS^LENGTH);
IF ERROR <> 0 THEN ...                      !Error condition
```

Resolving Subvolume Names

You can treat the input string to the FILENAME_RESOLVE_ procedure as a subvolume name by setting the *`subvol-resolve`* flag (bit 14) in the *`options`* parameter to 1.

The following example checks the output of the FILENAME_SCAN_ procedure to see whether it refers to a subvolume name. If so, the example sets the *`subvol-resolve`* flag to 1 before calling FILENAME_RESOLVE_:

```
LITERAL MAXLEN         = 256;
LITERAL ACCEPT^SUBVOLS = %B0000000000000010;
LITERAL RESOLVE^SUBVOL = %B0000000000000010;
  .
  .
OPTIONS := ACCEPT^SUBVOLS;
ERROR := FILENAME_SCAN_(STRING^BUFFER:STRING^LENGTH,
                        COUNT,
                        KIND,
                        LEVEL,
                        OPTIONS);
IF ERROR <> 0 THEN ...                    ! Error condition
ELSE
BEGIN
   IF LEVEL = 1
   THEN OPTIONS := RESOLVE^SUBVOL
   ELSE OPTIONS := 0;
   ERROR := FILENAME_RESOLVE_(STRING^BUFFER:COUNT,
                              FULLNAME:MAXLEN,
                              FULL^LENGTH,
                              OPTIONS);
   IF ERROR <> 0 THEN ...                 ! Error condition
END;
```

Resolving DEFINE Names

The FILENAME_RESOLVE_ procedure does not normally modify DEFINE names. If you supply a DEFINE name to this procedure, then the return string is usually the same as the input string but with all uppercase letters. However, you can perform some processing of DEFINEs by setting appropriate flags in the *options* parameter:

- *options* bit 12 is the *DEFINE-simple-resolve* flag, which resolves map DEFINEs
- *options* bit 11 is the *DEFINE-reduction* flag, which resolves DEFINEs that refer to a file name
- *options* bit 10 is the *DEFINE-reject* flag, which rejects DEFINEs that are not resolved to a file name

The following paragraphs describe these options in detail. For general information about DEFINEs, refer to Section 7, Using DEFINEs.

Resolving Map DEFINES

You can resolve a map DEFINE into the name contained in the DEFINE by setting the *DEFINE-simple-resolve* flag (bit 12) in the *options* parameter to 1 before calling the FILENAME_RESOLVE_ procedure. For any other class of DEFINE, the procedure returns the DEFINE name.

The *DEFINE-simple-resolve* flag also causes the system to check for the existence of the DEFINE. The FILENAME_RESOLVE_ procedure returns error 198 (unable to find DEFINE) if the DEFINE does not exist or error 13 (illegal file name) if DEFINE mode is not turned on.

The following example resolves map DEFINEs and checks for errors:

```
LITERAL DEFINE^SIMPLE^RESOLVE = %B0000000000001000;
  .
```

```
 .
OPTIONS := DEFINE^SIMPLE^RESOLVE;

ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           OPTIONS);
IF ERROR <> 0 THEN
CASE ERROR OF
BEGIN
   13 ->                   !DEFINE mode turned off
   198 ->                  !No such DEFINE
   OTHERWISE ->            !Other error
END;
```

Resolving DEFINEs That Contain a File Name

Tape, spool, and map class DEFINEs that refer to file names can be reduced to the file name that they refer to by setting the *`DEFINE-reduction`* flag (bit 11) in the *`options`* parameter to 1. All other information contained in the DEFINE is unavailable when the file name is used.

The *`DEFINE-reduction`* flag causes the FILENAME_RESOLVE_ procedure to return the name of the file contained in the DEFINE (if there is one) or the DEFINE name if there is none. If the DEFINE does not exist, then the procedure returns error 198. If DEFINE mode is turned off, then error 13 is returned.

The following example returns the file name referred to in a tape, spool, or map DEFINE.

```
LITERAL DEFINE^REDUCTION = %B0000000000010000;
 .
 .
OPTIONS := DEFINE^REDUCTION;
ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           OPTIONS);
IF ERROR <> 0 THEN
CASE ERROR OF
BEGIN
   13 ->                   !DEFINE mode turned off
   198 ->                  !No such DEFINE
   OTHERWISE ->            !Other error
END;
```

Note that sort, catalog, defaults, and search DEFINEs cannot be resolved by this option.

Rejecting DEFINEs That are not Resolved to a File Name

You have the option to reject any DEFINEs that are not resolved to a file name. You set the `DEFINE-reject` flag (bit 10) in the `options` parameter to 1 to request this feature. Instead of returning the name of such a DEFINE, FILENAME_RESOLVE_ returns error 13. This option can be used alone or with the preceding options.

The following example modifies the example given for the `DEFINE-reduction` flag by rejecting DEFINEs that do not reference a file name:

```
LITERAL DEFINE^REDUCTION = %B0000000000010000;
LITERAL DEFINE^REJECT    = %B0000000000100000;
  .
  .
OPTIONS := DEFINE^REDUCTION LOR DEFINE^REJECT;

ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,OPTIONS);
IF ERROR <> 0 THEN
CASE ERROR OF
BEGIN
   13 ->                   !DEFINE mode turned off, or
                           ! no file name referenced
   198 ->                  !No such DEFINE
   OTHERWISE ->            !Other error
END;
```

Searching and Resolving File Names

You can perform file-name resolution by searching a list of subvolumes contained in a search DEFINE. You specify the DEFINE in the `search` parameter of the FILENAME_RESOLVE_ procedure. Refer to Section 7, Using DEFINEs, for a description of search DEFINEs.

If the specified search DEFINE exists and DEFINE mode is turned on, then the system searches the subvolume list contained in the DEFINE for the file named in the input string. Note that searching is done only if the input string contains only the file ID (last part) of a file name.

The search proceeds as follows. The system searches the first subvolume listed in the search DEFINE. If a match is found, then the file name is resolved using that subvolume. If no match is found, the search continues with the next listed subvolume. If the search finishes without finding a match, error 11 (file not in directory) is normally returned.

The search is skipped without returning an error if one of the following conditions is true:

- The input string does not contain just a valid file ID.
- The search DEFINE length is zero.
- The search DEFINE does not exist.

- DEFINE mode is turned off.

The following example searches the subvolume list in a search DEFINE named =FINDIT. It looks for a file whose file ID is PROGA:

```
NAME ':=' "PROGA" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @NAME;
SEARCH^DEFINE ':=' "=FINDIT" -> @S^PTR;
S^DEFINE^LENGTH := @S^PTR '-' @SEARCH^DEFINE;
ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           !options!,
                           !override^name:length!,
                           SEARCH^DEFINE:S^DEFINE^LENGTH);
IF ERROR <> 0 THEN
CASE ERROR OF
BEGIN
   11 ->                   !File not found
    .
    .
   OTHERWISE ->            !Other error
END;
```

You can force file-name resolution even though the search failed to find a match by setting the *search-fail-OK* flag (bit 9) in the *options* parameter to 1 before calling the FILENAME_RESOLVE_ procedure. The file ID will be qualified by the first subvolume in the search DEFINE if no match is found.

Overriding the Input File Name With a DEFINE

Your program can give the user the ability to override the file name specified in the input string by supplying the name of a DEFINE that contains an override file name. To use this feature, your program must set the *override* parameter of the FILENAME_RESOLVE_ procedure to the name of the DEFINE that provides the override file name.

In the following example, the override name identifies a DEFINE named =MYDEFINE. If the DEFINE exists, then the file named in the DEFINE overrides the file name supplied in the input string in NAME. If the DEFINE does not exist, the input string in NAME is used as in the normal case.

```
NAME ':=' "PROGA" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @NAME;
OVERRIDE^NAME ':=' "=MYDEFINE" -> @S^PTR;
ORIDE^NAME^LENGTH := @S^PTR '-' @OVERRIDE^NAME;
ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           !options!,
                           OVERRIDE^NAME:ORIDE^NAME^LENGTH);
IF ERROR <> 0 THEN ...                       !Error condition
```

An alternative way to specify the override name is to use a map DEFINE with the same name as the file ID in the input string, prefixed with an equal sign (=). You can do this by setting the *`automatic-override`* flag (bit 8) in the *`options`* parameter to 1 before calling the FILENAME_RESOLVE_ procedure.

The following example resolves the file name in a DEFINE called =PROGA. If there is no such DEFINE, the input string PROGA is used:

```
LITERAL AUTO^OVERRIDE = %B0000000010000000;
  .
  .
NAME ':=' "PROGA" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @NAME;
OPTIONS := AUTO^OVERRIDE;
ERROR := FILENAME_RESOLVE_(NAME:NAME^LENGTH,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           OPTIONS);
IF ERROR <> 0 THEN ...                      !Error condition
```

Truncating Default Parts of File Names

To truncate the applicable default values for node name, volume name, and subvolume name from a file name, use the FILENAME_UNRESOLVE_ procedure. You may want to do this, for example, before displaying file names to local users.

The default values used by the FILENAME_UNRESOLVE_ procedure may be all or some of the current default values specified in the =_DEFAULTS DEFINE or they may be specified in an alternate defaults DEFINE. The following paragraphs describe these options.

△ **Caution.** Passing an invalid file name or file-name pattern to the FILENAME_UNRESOLVE_ procedure can result in a signal, trap, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

Truncating All Current Default Values

The file-name elements removed by the FILENAME_UNRESOLVE_ procedure are normally those that compare with the values set up in the =_DEFAULTS DEFINE.

The following example removes from a file name all elements that match the current default. The file name to be unresolved is passed to the procedure in FNAME and is returned, stripped of the default values, in SHORT^NAME:

```
LITERAL MAXLEN = 256;
  .
  .
ERROR := FILENAME_UNRESOLVE_(FNAME:LENGTH,
                             SHORT^NAME:MAXLEN,
                             SHORT^NAME^LENGTH);
IF ERROR <> 0 THEN ...                        !Error condition
```

Truncating a Specified Subset of the Default Values

Alternatively, you can request that all default values to the left of a specified file-name part be removed from the file name. Here, you need to use the *`level`* parameter to specify the level of the first part of the name that will not be removed.

The following example selects the device (or process) level. Here, the node name will be truncated from the file name if it matches the default node name. A device or subvolume name, however, will not be truncated, even if it matches the default value:

```
LITERAL MAXLEN       = 256;
LITERAL DEVICE^LEVEL =   0;
 .
 .
ERROR := FILENAME_UNRESOLVE_(FNAME:LENGTH,
                             SHORT^NAME:MAXLEN,
                             SHORT^NAME^LENGTH,
                             DEVICE^LEVEL);
IF ERROR <> 0 THEN ...                             !Error condition
```

The *`level`* parameter can have one of the following values:

-1 The first part always returned is the node name.

0 The first part always returned is the device name, process name, or logical device name.

1 The first part always returned is the first qualifier of the file name. For permanent disk files, this is the subvolume name.

2 This value refers to the second qualifier (or file ID for permanent disk files). No default values are returned.

Truncating Alternate Default Values

Normally, the FILENAME_UNRESOLVE_ procedure compares parts of the input file string with the default values specified in the =_DEFAULTS DEFINE. However, you can specify alternate default values using the *`defaults`* parameter.

The *`defaults`* parameter can name the subvolume (and optionally the volume and network node) directly or provide the name of a defaults DEFINE that contains the alternate names.

The following example specifies alternate default values directly:

```
LITERAL MAXLEN       = 256;
LITERAL DEVICE^LEVEL =   0;
 .
 .
ALT^DEFAULTS ':=' "\SYSA.$ARCHIVE.AUGUST";
ERROR := FILENAME_UNRESOLVE_(FNAME:LENGTH,
                             SHORT^NAME:MAXLEN,
                             SHORT^NAME^LENGTH,
                             DEVICE^LEVEL,
                             ALT^DEFAULTS);
IF ERROR <> 0 THEN ...                      !Error condition
```

Truncating Default Parts of File Names: Some Examples

The following examples show how the FILENAME_UNRESOLVE_ procedure deals with file names, given that the default values are \SYSA.$OURVOL.MYSUB:

Input File Name	**Level**	**Output File Name**
\SYSA.$OURVOL.MYSUB.PROGB	0	$OURVOL.MYSUB.PROGB
\SYSB.$YOURVOL.HISSUB.FILEA	0	\SYSB.$YOURVOL.HISSUB.FILEA
\SYSA.$THEIRVOL.HERSUB.FILEB	0	$THEIRVOL.HERSUB.FILEB
\SYSA.$OURVOL.RECORDS.LOGFILE	2	RECORDS.LOGFILE
RECORDS.LOGFILE	0	RECORDS.LOGFILE
MYSUB.PROGA	1	MYSUB.PROGA
MYSUB.PROGA	2	PROGA

Extracting Pieces of File Names

To extract pieces of file names, use the FILENAME_DECOMPOSE_ procedure. You pass the file name to the procedure along with an indication of the piece of the file name that you want to extract. The procedure returns the extracted piece. With normal use, a partially qualified file name is implicitly resolved; parts of the file name not specified in the input string can therefore be returned using the default values.

You can use the FILENAME_DECOMPOSE_ procedure to extract a single part of a file name, a file-name suffix, a file-name prefix, or a subpart of a process descriptor. Although file names are normally implicitly qualified, you can choose to extract file-name pieces without implicit resolution. The following paragraphs describe these options.

△ **Caution.** Passing an invalid file name or file-name pattern to the FILENAME_DECOMPOSE_ procedure can result in a signal, trap, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

Extracting a File-Name Part

To extract one part of a file name, you need to supply the FILENAME_DECOMPOSE_ procedure with the file name and the level of the part you want to extract. No additional options are necessary.

The following example extracts the subvolume name from a file name:

```
LITERAL MAXLEN       = 16;
LITERAL SUBVOL^LEVEL =  1;
  .
  .
```

```
ERROR := FILENAME_DECOMPOSE_(FNAME:LENGTH,
                             PART:MAXLEN,
                             PART^LENGTH,
                             SUBVOL^LEVEL);
IF ERROR <> 0 THEN ...                            !Error condition
```

Here, the file name is passed in FNAME. The subvolume name is returned in PART and its length in PART^LENGTH. MAXLEN is set to 16 to allow for the maximum size of a subvolume pattern.

The part that you want returned is specified in the *level* parameter (SUBVOL^LEVEL in the previous example). It can have one of the following values:

-1	Returns the node name.
0	Returns the device name, process name, or logical device name.
1	Returns the subvolume name for permanent disk files or returns the temporary file ID for temporary files.
2	Returns the file ID for permanent disk files.

Extracting a File-Name Suffix or a File-Name Prefix

In addition to returning the requested element, you can have the FILENAME_DECOMPOSE_ procedure return all elements to the right of the requested element by setting the *extract-suffix* flag (bit 15) in the *options* parameter to 1. Similarly, you can return all elements to the left of the selected element as well as the selected element itself by setting the *extract-prefix* flag (bit 14) in the *options* parameter to 1 before calling the FILENAME_DECOMPOSE_ procedure.

The following example extracts a file-name suffix, the first part of which is the subvolume name:

```
LITERAL MAXLEN         = 32;
LITERAL SUBVOL^LEVEL   = 2;
LITERAL EXTRACT^SUFFIX = %B0000000000000001;
 .
 .
OPTIONS := EXTRACT^SUFFIX;
ERROR := FILENAME_DECOMPOSE_(FNAME:LENGTH,
                             SUFFIX^PIECE:MAXLEN,
                             SUFFIX^LENGTH,
                             SUBVOL^LEVEL,OPTIONS);
IF ERROR <> 0 THEN ...                           !Error condition
```

Extracting File-Name Pieces Without Implicit Resolution

You can exclude all default values from the returned shortened name by setting the *no-defaults* flag (bit 13) in the *options* parameter to 1 before calling the FILENAME_DECOMPOSE_ procedure. In other words, setting the *no-defaults* flag turns off implicit resolution of partially qualified file names.

If, for example, the input string is “$OURVOL.MYSUB.PROGA” and you want the file-name prefix returned up to and including the subvolume, then “$OURVOL.MYSUB” is returned. The node name, although specified in the default values, is not returned.

The code to execute this example is shown below:

```
LITERAL MAXLEN         = 256;
LITERAL SUBVOL^LEVEL   = 2;
LITERAL EXTRACT^PREFIX = %B0000000000000010;
LITERAL NO^DEFAULTS    = %B0000000000000100;
 .
 .
OPTIONS := EXTRACT^PREFIX LOR NO^DEFAULTS;
ERROR := FILENAME_DECOMPOSE_(FNAME:LENGTH,
                         PREFIX^PIECE:MAXLEN,
                         PREFIX^LENGTH,
                         SUBVOL^LEVEL,OPTIONS);
IF ERROR <> 0 THEN ...                   !Error condition
```

Extracting Subparts of a Process Descriptor

If the name you are decomposing is a process descriptor, then you can divide the name further to extract subparts of the process name or identifying part of an unnamed process. To do this, you must include the *subpart* parameter in the procedure call. This parameter occurs at the end of the parameter list and can have any of the following values:

0 Return the entire element (the default action).

1 Return only the CPU part of an unnamed process-file name.

2 Return only the PIN.

3 Return only the process sequence number.

4 Return only the name subpart.

Values 1 and 2 apply only to unnamed processes. Value 4 applies only to named processes. Values 0 and 3 apply to named and unnamed processes.

The following example returns only the process sequence number in the variable SEQ^NUM:

```
LITERAL MAXLEN          = 16;
LITERAL PROCESS^NAME    = 0;
LITERAL EXTRACT^SEQ^NUM = 3;
 .
 .
LEVEL := PROCESS^NAME;
SUBPART := EXTRACT^SEQ^NUM;
ERROR := FILENAME_DECOMPOSE_(FNAME:LENGTH,
                             SEQ^NUM:MAXLEN,SEQ^NUM^LENGTH,
                             LEVEL,
                             !options!,
                             SUBPART);
IF ERROR <> 0 THEN ...                       !Error condition
```

Extracting Pieces of File Names: Some Examples

The following examples list some file names and the corresponding output from the FILENAME_DECOMPOSE_ procedure. The examples assume that the current default values are \SYS.$OURVOL.MYSUB:

Input Name	Level	Options	Subpart	Output Element(s)
$YOURVOL.HISSUB.FILEA	0			$YOURVOL
$YOURVOL.HISSUB.FILEA	0	suffix		$YOURVOL.HISSUB.FILEA
FILE1	0			$OURVOL
FILE1	0	suffix		$OURVOL.MYSUB.FILE1
FILE1	0	prefix		\SYS.$OURVOL
$P:4321.#A	0			$P:4321
$P:4321.#A	0		4	$P

Modifying Portions of a File Name

To modify a piece of a file name, use the FILENAME_EDIT_ procedure.

You must specify the piece of the file name you need to modify and the character string that replaces that piece. If the replacement string is zero length, then the piece is simply removed (leaving the correct number of part separator characters).

The input string contains the file name you need to modify. The name can be fully or partially qualified. If the name is partially qualified, then the system applies the default values from the =_DEFAULTS DEFINE. You can therefore edit any part of the fully qualified name, even if the input string contained a partially qualified name. An invalid input string might cause error 13 to be returned.

The piece of the file name that you replace can be a file-name part, a file-name suffix, a file-name prefix, or a subpart of a process descriptor. The following paragraphs show how.

△ **Caution.** Passing an invalid file name or file-name pattern to the FILENAME_EDIT_ procedure can result in a signal, trap, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

Modifying One Part of a File Name

Use the *level* parameter to specify the part of the file name you want to change. The following example replaces the volume name of the file name \SYSA.$YOURVOL.RECORDS.LOGFILE:

```
LITERAL MAXLEN       = 256;
LITERAL VOLUME^LEVEL =   0;
 .
 .
FNAME ':=' "\SYSA.$YOURVOL.RECORDS.LOGFILE" -> @S^PTR;
FNAME^LENGTH := @S^PTR '-' @FNAME;
NEW^PART ':=' "$OURVOL" -> @S^PTR;
PART^LENGTH := @S^PTR '-' @NEW^PART;
ERROR := FILENAME_EDIT_(FNAME:MAXLEN,
                        FNAME^LENGTH,
                        NEW^PART:PART^LENGTH,
                        VOLUME^LEVEL);
IF ERROR <> 0 THEN ...                       !Error condition
```

In the example above, the name to be changed and its length are passed to the FILENAME_EDIT_ procedure in FNAME and FNAME^LENGTH. The new value of the volume part of the name and its length are passed in NEW^PART:PART^LENGTH. The procedure uses this information to replace the volume part in the old file name because the volume level (level 0) is specified in the *level* parameter.

The edited file name is returned in FNAME and its length in FNAME^LENGTH.

Replacing a File-Name Suffix or File-Name Prefix

To replace a file-name suffix of more than one part, you need to set the *suffix* flag (bit 15) in the *options* parameter to 1. The *level* parameter identifies the start of the suffix. The supplied replacement string is substituted for the part specified by the *level* parameter and all parts to its right. Similarly, you can replace a file-name prefix by setting the *prefix* flag (bit 14) in the *options* parameter to 1. The *level* parameter identifies the last part of the prefix.

The following example changes the input file name from \SYSA.$OURVOL.RECORDS.LOGFILE to \SYSA.$OURVOL.RECORDS1.ARCHIVE. That is, the subvolume and file ID are replaced:

```
LITERAL MAXLEN       = 256;
LITERAL SUBVOL^LEVEL = 1;
LITERAL SUFFIX       = %B0000000000000001;
 .
 .
FNAME ':=' "\SYSA.$OURVOL.RECORDS.LOGFILE" -> @S^PTR;
FNAME^LENGTH := @S^PTR '-' @FNAME;
NEW^PIECE ':=' "RECORDS1.ARCHIVE" -> @S^PTR;
PIECE^LENGTH := @S^PTR '-' @NEW^PIECE;
ERROR := FILENAME_EDIT_(FNAME:MAXLEN,
                        FNAME^LENGTH,
                        NEW^PIECE:PIECE^LENGTH,
                                SUBVOL^LEVEL,
                                SUFFIX);
IF ERROR <> 0 THEN ...                              !Error condition
```

Replacing a Subpart of a Process ID

To replace any subpart of a process ID, you need to use the *`subpart`* parameter of the FILENAME_EDIT_ procedure. The *`subpart`* parameter specifies which element of the process identifier you intend to change. For named processes, you can modify the process name or its sequence number. For unnamed processes, you can modify the CPU number, PIN, or sequence number.

You set the *`subpart`* parameter according to the subpart you intend to replace as follows:

0	Replace the entire element (the default action).
1	Replace only the CPU part of an unnamed process file name.
2	Replace only the PIN.
3	Replace only the process sequence number.
4	Replace only the name part.

The following example changes the name of the named process \SYSA.$P1:321 to \SYSA.$P2:321:

```
LITERAL MAXLEN               = 256;
LITERAL PROCESS^LEVEL        =   0;
LITERAL PROCESS^NAME^SUBPART =   4;
  .
  .
FNAME ':=' "\SYSA.$P1:321" -> @S^PTR;
FNAME^LENGTH := @S^PTR '-' @FNAME;
NEW^SUBPART ':=' "$P2" -> @S^PTR;
NEW^SUBPART^LENGTH := @S^PTR '-' @NEW^SUBPART;
ERROR := FILENAME_EDIT_(FNAME:MAXLEN,FNAME^LENGTH,
                        NEW^SUBPART:NEW^SUBPART^LENGTH,
                        PROCESS^LEVEL,
                        !options!,
                        PROCESS^NAME^SUBPART);
IF ERROR <> 0 THEN ...                             !Error condition
```

Comparing File Names

To compare two file names, use the FILENAME_COMPARE_ procedure. This procedure returns either 0 if two names refer to the same object or -1 if the names differ.

The following example compares a permanent disk-file name with a map DEFINE name:

```
FNAME1 ':=' "\SYSA.$OURVOL.MYSUB.PROGA" -> @S^PTR;
FNAME1^LENGTH := @S^PTR '-' @FNAME1;
FNAME2 ':=' "=MYPROG" -> S^PTR;
FNAME2^LENGTH := @S^PTR '-' @FNAME2;
STATUS := FILENAME_COMPARE_(FNAME1:FNAME1^LENGTH,
                            FNAME2:FNAME2^LENGTH);
```

The procedure accepts partially qualified file names and implicitly expands them to their fully qualified form before comparing.

△ **Caution.** Passing an invalid file name to the FILENAME_COMPARE_ procedure can result in a signal, trap, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

Searching For and Matching File-Name Patterns

You can use file-name patterns to search for files. For example, you may want a list of all disk files on your network whose names begin with the letter Z. To do this, you start a search for the file-name pattern *.*.*.Z*.

A search always involves the following procedure calls:

- FILENAME_FINDSTART_ establishes the start of a search by providing the file-name pattern to search for.
- FILENAME_FINDNEXT_ is usually called repeatedly. On each call, this procedure finds the next file name that matches the pattern established by FILENAME_FINDSTART_.
- FILENAME_FINDFINISH_ releases resources used by the search. This procedure is called when the search is complete.

This subsection describes how to use these procedures. The sample program at the end of this section includes a procedure that searches for file-name patterns.

In addition to the system procedures listed above, this subsection also describes how you can match a process qualifier string with a file-name pattern using the FILENAME_MATCH_ procedure.

Establishing the Start of a File-Name Search

Use FILENAME_FINDSTART_ to set up a search for file names. You can search for systems, devices, and named processes, or subvolumes, files, and subdevices.

In addition to setting up a pattern to search for, the FILENAME_FINDSTART_ procedure has several options that allow you to do the following: specify the level at which the subsequent search reports file names, limit a search to device files only, make special provisions when searching for process names, set up an asynchronous search, and report specific kinds of system or device errors encountered during a search. The following paragraphs describe these options.

Specifying the Search Pattern

To use the FILENAME_FINDSTART_ procedure, you must pass to it the file name pattern to search for, along with its length. The procedure returns a search ID that you use to identify this search to the FILENAME_FINDNEXT_ and FILENAME_FINDFINISH_ procedures. This method allows you to have up to 16 searches concurrently active.

The following example sets up a search for a file named PROGA in any subvolume in the current default volume:

```
SEARCH^PATTERN ':=' "*.PROGA" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH);
IF ERROR <> 0 THEN ...                          !Error condition
```

Setting the Resolution Level

You can specify the level at which file names are reported by the subsequent search. To do this, you include the *resolvelevel* parameter in the FILENAME_FINDSTART_ procedure call. You set this value to the desired level as follows:

-1	Specifies the node name level
0	Specifies the device, process, or logical device level
1	Specifies the first qualifier (subvolume for a disk device)
2	Specifies the second qualifier (file ID for a disk device)

If, for example, the *resolvelevel* parameter is set to 0, then all file names found in the subsequent search are resolved to the device, process, or logical device level. That is, the resolved file names will not include the node name:

```
SEARCH^PATTERN ':=' "*.PROGA" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
RESOLVE^LEVEL := 0;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             RESOLVE^LEVEL);
IF ERROR <> 0 THEN ...                          !Error condition
```

Note that the resolve level must not be to the right of a part of the search pattern that contains a wild-card character or error 590 (invalid parameter) occurs. For example, if the resolve level is zero, then the search pattern must not contain wild-card characters in the node name.

Setting Up a Search for a Specific Type of Device

The FILENAME_FINDSTART_ procedure allows you to restrict the output of a search to files of a specified device type or subdevice type. In addition, if you set the *`not-device-type`* flag (bit 14) or the *`not-subdevice-type`* flag (bit 13) in the *`options`* parameter, you can restrict the report to all but the specified device type and subdevice type.

You specify the device type you want in the *`devtype`* parameter. You specify the subdevice in the *`subdevtype`* parameter.

These device-restricting options are most useful when restricting searches to disk files. However, these options can also be used for devices other than disks, but with restrictions as described below in Searching for Files Not on Disk. The following paragraphs describe how to use the device-restricting options.

Searching for Disk Files

The recommended use of the device-restricting options is to limit a search to disk-file names. You can significantly reduce the search time by not attempting to match a pattern with the names of files not on disk if you know that the files you are searching for are disk files.

By setting the *`devtype`* parameter to 3, you restrict the search to disk files:

```
SEARCH^PATTERN ':=' "$*.*.PROGA" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
RESOLVE^LEVEL := 0;
DEVICE^TYPE := 3;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             RESOLVE^LEVEL,DEVICE^TYPE);
IF ERROR <> 0 THEN ...                      !Error condition
```

Searching for Files Not on Disk

Searching for files that are not on disk is more complex because devices other than disks can have subdevices with device types that are different from their parent device. In addition, a process that simulates a device type may have process qualifiers representing various device types. Note that neither of these restrictions apply to disk files; disk files and subvolumes always have the same device type as their parent volume, and, moreover, a process cannot simulate a disk device.

The implication of this is that you do not significantly reduce the search time by restricting a search to a device type other than disk (although the list of matching files may be shortened).

You can, however, eliminate the problem of processes that simulate devices by avoiding searching for them. To do so, you need to set the *`no-device-simulation`* flag (bit 10) in the *`options`* parameter to 1. These processes will then be regarded as subtype 30 processes rather than the device type of the devices they simulate.

Setting Up a Search for Process Qualifier Names

For the qualifier names of a process to be seen by the procedures that search for file names, the process must indicate its ability to perform qualifier name searches. It does so by issuing a PROCESS_SETINFO_ procedure call as follows:

```
LITERAL QUALIFIER^INFO = 49;
  .
  .
ATTVAL := 1;
ERROR := PROCESS_SETINFO_(!process^handle!,
                          !specifier!,
                          QUALIFIER^INFO,ATTVAL,1);
```

A process which does this must be prepared to service -107 system messages arriving on $RECEIVE.

For processes that make their qualifiers known in this way, you can bypass process qualifiers when searching for names. You do so by setting the *`no-subprocesses`* flag (bit 11) in the *`options`* parameter to 1:

```
LITERAL NO^SUBPROCESSES = %B0000000000010000;
  .
  .
SEARCH^PATTERN ':=' "$L*.#*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
RESOLVE^LEVEL := 0;
OPTIONS := NO^SUBPROCESSES;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             RESOLVE^LEVEL,
                             !device^type!,
                             !device^subtype!,
                             OPTIONS);
IF ERROR <> 0 THEN ...                          !Error condition
```

Specifying a Name to Start Searching From

You can select a name at which the search will start. This feature might be useful, for example, when restarting a search that has been interrupted.

You specify the name in the *`startname`* parameter of the FILENAME_FINDSTART_ procedure. This name should be somewhere in the sequence of names described by the file-name pattern.

Normally, the search starts at the named file or at the next name in the sequence if the named file does not exist. However, you can force the search to start at the name following the named file, even if the name does exist, by setting the *`skip-if-same`* flag (bit 15) in the *`options`* parameter to 1.

The following example starts a search at the file following \SYSB.$ARCHIVE.S110189 for the file-name pattern *.*.*.*. The search is limited to disk files:

```
LITERAL SKIP^IF^SAME = %B0000000000000001;
 .
 .
SEARCH^PATTERN ':=' "\*.*.*.*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
DEVICE^TYPE := 3;
OPTIONS := SKIP^IF^SAME;
START^NAME ':=' "\SYSB.$ARCHIVE.S110189" -> @S^PTR;
S^NAME^LENGTH := @S^PTR '-' @START^NAME;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             !resolve^level!,
                             DEVICE^TYPE,
                             !device^subtype!,
                             OPTIONS,
                             START^NAME:S^NAME^LENGTH);
IF ERROR <> 0 THEN ...                      !Error condition
```

Note. For some file types, the search sequence might be alphabetic; for other file types, it might not be. However, for a given file type, the search sequence is always the same for the same release of the operating system.

Reporting Device or System Failures

You can choose to receive notification of failed or offline devices and systems encountered during a search. To be sure that such errors are always reported, set the *report-off-line* flag (bit 12) in the *options* parameter to 1 before calling FILENAME_FINDSTART_.

The following example sets the *report-off-line* flag to 1:

```
LITERAL REPORT^OFFLINE = %B0000000000001000;
 .
 .
SEARCH^PATTERN ':=' "\*.*.*.*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
OPTIONS := REPORT^OFFLINE;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             !resolve^level!,
                             !device^type!,
                             !device^subtype!,
                             OPTIONS);
IF ERROR <> 0 THEN ...                        !Error condition
```

Errors that are reported when the *report-off-line* flag is 1 that might otherwise not be reported include:

Errors 62 to 66	Device off line
Error 250	System not connected

If the *report-off-line* flag is zero, then devices or systems that are offline or in a failed state are skipped over when encountered in a search if the device or node is specified generically (that is, if the device or node part of the file name contains a wild-card character or if the piece of the file-name pattern to the left of the device name contains a wild-card character). For example:

\SYSA.$OURVOL.*	The node name and volume name are specified explicitly. Neither is generic.
\SYSA.$*.*	The node name is explicit, but the volume (or process) name is generic.
*.$OURVOL	The node name is generic and the volume name is generic because the node name is.

When you explicitly specify a device in a file-name pattern, the system always reports device errors whether the *report-off-line* flag is set or not.

Finding the Next Matching File Name

After setting up a search using the FILENAME_FINDSTART_ procedure, you can search for the specified file-name pattern using calls to the FILENAME_FINDNEXT_ procedure.

The FILENAME_FINDNEXT_ procedure requires the search ID returned by the FILENAME_FINDSTART_ procedure. From this parameter, the FILENAME_FINDNEXT_ procedure can derive the pattern to search for.

The FILENAME_FINDNEXT_ procedure normally performs a waited search. If the search was set up nowait, then the search proceeds asynchronously. The following paragraphs describe how to program for both of these situations, as well as how you can get file-characteristic information about the returned named entities and how to handle some system errors that could occur during searching.

Performing a Waited Search

When the FILENAME_FINDNEXT_ procedure finds a match, it returns the name found in its *name* parameter. Also, following a successful search, the *error* returned is 0. If the system cannot find a matching name, then the *error* returned is 1.

The following example sets up a search for all files named PROGA on any subvolume of the current volume of the current system. The search ID returned by the FILENAME_FINDSTART_ procedure identifies the search to the FILENAME_FINDNEXT_ procedure, which returns in NAME the first name that matches the pattern:

```
SEARCH^PATTERN ':=' "*.PROGA" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH);
IF ERROR <> 0 THEN ...                          !Error condition

ERROR := FILENAME_FINDNEXT_(SEARCH^ID,
                            NAME:MAXLEN,
                            NAMELEN);
IF ERROR <> 0 THEN ...                     !Error condition
```

Note. The sequence in which names are returned by repeated calls to the FILENAME_FINDNEXT_ procedure depends on the subsystem. The sequence might not be in alphabetical order.

Performing a Nowait Search

To avoid having to wait for a user process to respond to a file-name search request, you can search in a nowait manner. To do so, you must set the *nowait* flag (bit 9) in the *options* parameter to 1.

The FILENAME_FINDNEXT_ procedure normally returns file names in a synchronous way. By specifying the *nowait* option, your process can continue while the search for the next match continues asynchronously. Instead of returning the found file name in the FILENAME_FINDNEXT_ parameter, however, the file name is returned in a message in $RECEIVE.

The returned message is system message -109 (Nowait FILENAME_FINDNEXT_ completion). The name of the returned entity starts in word 14 and has a length in bytes given by the value in word 8. If the search returns an error, the error number is returned in word 2.

If your program is running multiple concurrent searches, then you will also need to set the *tag* parameter in the FILENAME_FINDNEXT_ procedure call. You can then check which search is finishing by comparing words 9 and 10 of the FILENAME_FINDNEXT_ completion message with the tag supplied in the procedure call.

The following example performs asynchronous searching:

```
LITERAL NOWAIT = %B0000000001000000;

!Open $RECEIVE:
FILE^NAME ':=' "$RECEIVE" -> @S^PTR;
ERROR := FILE_OPEN_(FILE^NAME:@FILE^NAME '-' @S^PTR,
                    RECV^NUM);
IF ERROR <> 0 THEN ...

!Set up the search:
SEARCH^PATTERN ':=' "$*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
OPTIONS := NOWAIT;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH,
                             !resolve^level!,
                             !device^type!,
                             !device^subtype!,
                             OPTIONS);
IF ERROR = 1 THEN ...                        !No match found
ELSE IF ERROR <> 0 THEN ...                  !Error condition

!Start searching:
ERROR := FILENAME_FINDNEXT_(SEARCH^ID);
IF ERROR <> 0 THEN ...
 .
 .

!Continue processing asynchronously
 .
 .

!Read $RECEIVE
READUPDATEX(RECV^NUM,SBUFFER,
           READ^COUNT);
IF <> THEN
BEGIN

   !Check if system message:
   CALL FILE_GETINFO_(RECV^NUM,
                      ERROR);
   IF ERROR = 6 THEN
   BEGIN

      !Continue processing based on message number:
      CASE BUFFER[0] OF
      BEGIN

         !If message is nowait return from
         !FILENAME_FINDNEXT_, check word 2 of message for
         !search error.  If no error, move the name string
         !out of the message and into the NAME variable:
         -109 -> BEGIN
            IF BUFFER[2] = 0 THEN
               NAME ':=' BUFFER[14] FOR BUFFER[8];
         END;
          .
          .!Other system messages:

         OTHERWISE ->...
      END;
   END

   !Or if it is not a system message:
   ELSE ...
END;
```

For complete details on the Nowait FILENAME_FINDNEXT_ completion message, refer to the *Guardian Procedure Errors and Messages Manual*.

Returning Characteristics of Found Entities

You can retrieve information about each entity returned by the FILENAME_FINDNEXT_ procedure by supplying the *`entityinfo`* parameter—a container for the returned information. Returning information in this way is often more convenient than calling other procedures to retrieve the same information.

Information returned in the *`entityinfo`* parameter includes the following:

Word 0	Contains the device type of the entity. Device types are listed in the *Guardian Procedure Calls Reference Manual*.
Word 1	Contains the device subtype of the entity. Device subtypes are listed in the *Guardian Procedure Calls Reference Manual*.

Word 2	For disks, contains the object type. If > 0, then the returned name refers to an SQL object type. If 0, then the returned name refers to a non-SQL file. If -1, then the returned name refers to a subvolume or volume.
Word 3	For a disk file, contains the file type: 0 for an unstructured file 1 for a relative file 2 for an entry-sequenced file 3 for a key-sequenced file -1 for a volume or subvolume
Word 4	For a disk file, contains the file code given to the file. For a subvolume or volume, this word contains -1.

For files that are not disk files, words 2, 3, and 4 are undefined.

The following example uses the *entityinfo* parameter to determine whether a returned entity is a temporary file name or a subvolume name (they both have the same format). Word 2 of the *entityinfo* parameter is -1 for a subvolume but will have some other value for a disk file:

```
SEARCH^PATTERN ':=' "$OURVOL.*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @SEARCH^PATTERN;
ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                             SEARCH^PATTERN:PATTERN^LENGTH);
IF ERROR <> 0 THEN ...                      !Error condition

ERROR := FILENAME_FINDNEXT_(SEARCH^ID,
                            NAME:MAXLEN,
                            NAMELEN,
                            ENTITY^INFO);
IF ERROR = 1 THEN ...                       !No match found
ELSE IF ERROR <> 0 THEN ...                 !Error condition

IF ENTITY^INFO[2] = -1 THEN ...             !subvolume
ELSE IF ENTITY^INFO[2] <> -1 THEN...        !temporary file
```

Handling Search Errors

For a waited search, errors are returned in the *error* variable. For a nowait search, errors may be returned either in the error variable when the search is initiated or in word 2 of the Nowait FILENAME_FINDNEXT_ completion message when the search finishes.

If generic offline errors are reported (see Establishing the Start of a File-Name Search earlier in this section), you can still continue with the search. You can recognize these errors by the fact that, even though an error is returned, a name is also returned. For these errors, the name is that of the entity (node or device) associated with the error and may be a name that is not in the form being searched for. You can use this name for error reporting.

If you continue searching from one of these errors by issuing another call to the FILENAME_FINDNEXT_ procedure, the set of names subordinate to the entity in error is skipped and the search continues with the next entity at the same level as the erroneous entity.

For errors that do not return a name, it is generally not worth retrying the search because the condition causing the error is likely to recur.

Terminating the File-Name Search

Once you have completed a file-name search, you should release the system resources allocated to the search. You do this by issuing a call to the FILENAME_FINDFINISH_ procedure.

To identify the correct search to terminate, you must supply the FILENAME_FINDFINISH_ procedure with the search ID that was returned by the corresponding call to FILENAME_FINDSTART_:

```
ERROR := FILENAME_FINDFINISH_(SEARCH^ID);
IF ERROR <> 0 THEN ...                       !Error condition
```

File-Name Matching

To check qualifier strings against a file-name pattern, you can use the FILENAME_MATCH_ procedure. This procedure can be used for process names and file names, so long as they are fully qualified. The intent of this feature is to enable you to support the use of wild-card characters on the qualifier names provided by your processes to other users.

The result of a FILENAME_MATCH_ procedure call can indicate a complete match or an incomplete match. The following paragraphs describe these outcomes.

Testing for a Complete Match

You provide the FILENAME_MATCH_ procedure with a pattern, a name, and their corresponding lengths. Both the pattern and the name must have the same level of left-hand qualification. A name containing a node name therefore matches only a pattern that also has a node-name part. (Note that you can expand all names and patterns to their fully qualified form using the FILENAME_RESOLVE_ procedure.)

The output of the FILENAME_MATCH_ procedure is a *`status`* value that simply indicates whether the file name matches the pattern. For a complete match, the procedure returns 2 in the status variable. A value of 0 indicates no match. A value of 1 indicates an incomplete match (see below), and a value of less than zero indicates an error.

The following example checks for a complete match:

```
STATUS := FILENAME_MATCH_(FULL^NAME:NAME^LENGTH,
                          PATTERN:PATTERN^LENGTH);
CASE STATUS OF
```

```
BEGIN
   2 ->                                !Complete match
   1 ->                                !Incomplete match
   0 ->                                !No match
   OTHERWISE ->                        !Error
END;
```

Testing for an Incomplete Match

An incomplete match status value (1) is returned if the name under test matches the left-hand portion of a pattern but not the entire pattern. For example, if the name under test is \SYSA.$OURVOL and the pattern is *.*.*, then the procedure returns an incomplete match.

The incomplete match can be useful in eliminating needless name searching where large hierarchies are involved. For example, you can test the node name and volume name for an incomplete match before going on to test for a match at the process and process-qualifier level.

The following example extracts the volume prefix from a name and checks for an incomplete match with the file-name pattern. If the match is successful, then the code checks the entire string.

```
LITERAL MAXLEN := 256;
LITERAL VOLUME := 0;
LITERAL EXTRACT^PREFIX = %B0000000000000010;
 .
 .
!Scan the file name to check that it is valid:
ERROR := FILENAME_SCAN_(STRING:LENGTH,
                        NAME^LENGTH);
IF ERROR <> 0 THEN ...                   ! Error condition
ELSE
BEGIN

   !Expand the file name to its fully qualified form:
   ERROR := FILENAME_RESOLVE_(STRING:NAME^LENGTH,
                              FULLNAME:MAXLEN,
                              FULL^LENGTH);
   IF ERROR <> 0 THEN ...                !Error condition
END;


!Extract the volume-level prefix:
LEVEL := VOLUME;
OPTIONS := EXTRACT^PREFIX;
CALL FILENAME_DECOMPOSE_(FULLNAME:FULL^LENGTH,
                         PREFIX:MAXLEN,
                         PREFIX^LENGTH,
                         LEVEL,
                         OPTIONS);

PATTERN ':=' "\*.*.*.*" -> @S^PTR;
PATTERN^LENGTH := @S^PTR '-' @PATTERN;
```

```
!Check for an incomplete match between the volume level
!prefix and the complete file-name pattern:
STATUS := FILENAME_MATCH_(PREFIX:PREFIX^LENGTH,
                          PATTERN:PATTERN^LENGTH);

CASE STATUS OF
BEGIN

      !Incomplete match:
      1 -> BEGIN

         !Check for a complete match
         STATUS := FILENAME_MATCH_(FULLNAME:FULL^LENGTH,
                                   PATTERN:PATTERN^LENGTH);
         CASE STATUS OF
         BEGIN
            2 ->                           !Complete match
            0 ->                           !No match
            OTHERWISE ->                   !Error condition
         END;
      END;

      !No match:
      0 ->

      !Error:
      OTHERWISE ->
END;
```

Matching File Names: Some Examples

The following examples show the result of comparing a name with a name pattern using the FILENAME_MATCH_ procedure:

Name	**Pattern**	**Result**
$PROC1.#Q1	$P*1.*	Complete match
$PROC1	$P*1.*	Incomplete match
\SYSA.$PROC1.#Q1	$P*1.*	No match

Manipulating File Names: An Example

This subsection presents a sample program that lists file names, resolved to their fully qualified form. The program should be run from the TACL prompt. It expects one parameter that specifies the names to be listed.

The user specifies one of the following in the command-line parameter:

- A single file name. This name can be partially or fully qualified. In either case, the program displays the fully qualified name.

- A map DEFINE. The name contained in the map DEFINE is expanded to its fully qualified form and displayed.
- A name pattern. Every file name that matches the name pattern is displayed in its fully qualified form.

Because the program reads and processes the Startup message, the user can specify the input and output file names. For a detailed discussion of the Startup message, refer to Section 8, Communicating With a TACL Process.

Some sample executions and their results are shown below. These examples assume that the current default values are \SYSA.$OURVOL.MYSUB and that this subvolume contains the following files: PROGA, PROGB, PROGC, ZPROGA, ZPROGB, and ZPROGC:

Command Input	**Result**
> RESOLVE PROGA	\SYSA.$OURVOL.MYSUB.PROGA
> RESOLVE PROGZ	\SYSA.$OURVOL.MYSUB.PROGZ
> RESOLVE Z*	\SYSA.$OURVOL.MYSUB.ZPROGA \SYSA.$OURVOL.MYSUB.ZPROGB \SYSA.$OURVOL.MYSUB.ZPROGC
> RESOLVE $ARCHIVE.RECORDS.*	All file names in subvolume \SYSA.$ARCHIVE.RECORDS
> RESOLVE *.*.*.*	All permanent disk files and all processes with two qualifiers on all systems in the network
> RESOLVE =MYMAP	\SYSA.$OURVOL.MYSUB.ZPROGA

The last example assumes that =MYMAP is a currently active map DEFINE containing the file name ZPROGA.

The program is made up of the following procedures:

- The INITIAL procedure is the main procedure. INITIAL calls the INITIALIZER system procedure to read and process the Startup message and then processes the parameter string supplied in the Startup message. It scans and resolves the parameter string and then responds according to what the string contains:

  2 For a simple file name or a map DEFINE, it calls the PRINT^NAME procedure to print the name.

  2 For a name pattern, it calls the FIND^FILES procedure to process each file name that matches the pattern.

- The START^IT procedure is invoked through the INITIALIZER procedure to process the Startup message.
- The INIT procedure opens the output file and returns the file number.
- The PRINT^NAME procedure simply writes the name of a file to the output file.

- The FIND^FILES procedure searches for all file names that match a given pattern. It calls PRINT^NAME for each match that it finds.
- The FILE^ERRORS procedure reports file-system errors by sending the error number to the output file.

The TAL source code for this program follows.

```
? INSPECT, SYMBOLS, NOCODE, NOMAP
?NOLIST, SOURCE $TOOLS.ZTOOLD00.ZSYSTAL
?LIST

------------------------------------------------------------
!Global parameters
------------------------------------------------------------

LITERAL ACCEPT^PATTERNS =
        %B0000000000000001;        !for FILENAME_SCAN_
LITERAL DEFINE^SIMPLE^RESOLVE =
        %B0000000000001000;        !for FILENAME_RESOLVE_
LITERAL DEFINE^REJECT =
        %B0000000000100000;        !for FILENAME_RESOLVE_
LITERAL MAXLEN =
        ZSYS^VAL^LEN^FILENAME;     !maximum file-name length
LITERAL MAXPATTERN = 512;          !maximum pattern length
INT     ERROR;                     !error return
INT     OUTNUM;                    !OUT file number
INT     INNUM;                     !IN file number
STRING .S^PTR;                     !pointer to end of string

STRUCT .CI^STARTUP;                !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULTS;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAM[0:529];
END;
```

```
INT PARAM^LEN;                        !length of PARAM string

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  FILE_OPEN_,FILENAME_SCAN_,FILENAME_RESOLVE_,
?  FILENAME_FINDSTART_,FILENAME_FINDNEXT_,
?  FILENAME_FINDFINISH_,WRITEX,PROCESS_STOP_,DNUMOUT,
?  OLDFILENAME_TO_FILENAME_,FILE_GETINFO_)
? LIST

!------------------------------------------------------------
! These DEFINEs help to format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N) =
          @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!------------------------------------------------------------
! Procedure to print file-system error numbers on the
! output file.
!------------------------------------------------------------

PROC FILE^ERRORS(ERROR);
INT ERROR;
BEGIN
   STRING .SBUFFER[0:36];             !output buffer

   START^LINE;
   PUT^STR ("File System Error Number Is:  ");

!  Write error to output file:

   PUT^INT (ERROR);
   CALL WRITEX(INNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Stop the process:

   CALL PROCESS_STOP_;
END;
```

```
!-------------------------------------------------------------
! Procedure to write a file name to the output file.
!-------------------------------------------------------------

PROC PRINT^NAME(NAME,LENGTH);
INT     LENGTH;
STRING .NAME;

BEGIN

   CALL WRITEX(OUTNUM,NAME,LENGTH);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(OUTNUM,ERROR);
      CALL FILE^ERRORS(ERROR);
   END;
END;


!-------------------------------------------------------------
! Procedure to save the Startup message in a global
! structure.
!-------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,MATCH) VARIABLE;
INT .RUCB,.START^DATA,.MESSAGE,LENGTH,MATCH;

BEGIN

   CI^STARTUP.MSGCODE ':=' MESSAGE FOR LENGTH/2;
   PARAM^LEN := LENGTH - 66;

END;
```

```
!-------------------------------------------------------------
! Procedure to perform initialization for the program.
!-------------------------------------------------------------

PROC INIT;

BEGIN
   STRING .OUT^NAME[0:MAXLEN - 1];   !string form of OUT file
                                     ! name
   INT     OUTNAME^LEN;              !length of OUT file

!  Call INITIALIZER to read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Convert the output file name:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.OUTFILE.VOLUME,
                        OUT^NAME:MAXLEN,
                        OUTNAME^LEN);
  IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);

!  Open the output file:

   ERROR := FILE_OPEN_(OUT^NAME:OUTNAME^LEN,OUTNUM);
   IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);

END;
```

```
!-------------------------------------------------------------
! Procedure to find all file names that match a given pattern
! and print each file name.
!-------------------------------------------------------------

PROC FIND^FILES(PATTERN,LENGTH);
INT    LENGTH;
STRING .PATTERN;

BEGIN
   INT SEARCH^ID;                      !identifies a search
   STRING .NAME[0:MAXLEN - 1];         !found file-name string
   INT NAMELEN;                        !length of found file
                                       ! name

!  Set up the search pattern:

   ERROR := FILENAME_FINDSTART_(SEARCH^ID,
                                PATTERN:LENGTH);
   IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);

!  Loop until pattern ranges exhausted:

   WHILE ERROR <> 1 DO
   BEGIN

   !  Find the next file name that matches pattern:

      ERROR := FILENAME_FINDNEXT_(SEARCH^ID,
                                  NAME:MAXLEN,
                                  NAMELEN);
      IF ERROR > 1 THEN CALL FILE^ERRORS(ERROR)

   !  Write matching file name to output file:

      ELSE CALL PRINT^NAME(NAME,NAMELEN);
   END;

!  Release resources held by search:

   ERROR := FILENAME_FINDFINISH_(SEARCH^ID);
   IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);
END;
```

```
!------------------------------------------------------------
! Main procedure determines kind of file name or pattern,
! resolves the name, and calls either PRINT^NAME for file
! names or DEFINEs or FIND^FILES for a file-name pattern.
!------------------------------------------------------------

PROC INITIAL MAIN;
BEGIN
   INT COUNT,KIND,LEVEL,OPTIONS;      !parameters for
                                      ! FILENAME_SCAN_
   INT FULL^LENGTH;                   !length of resolved
                                      ! file name or pattern
   STRING .FULLNAME [0:MAXLEN - 1];   !resolved file name
   STRING .PATTERN [0:MAXPATTERN - 1];!resolved name pattern

!  Read and save the Startup message and open the IN and OUT
!  files:

   CALL INIT;

!  Scan the file name or pattern returned from Startup
!  message:

   OPTIONS := ACCEPT^PATTERNS;
   ERROR := FILENAME_SCAN_(CI^STARTUP.PARAM:PARAM^LEN,
                           COUNT,
                           KIND,
                           LEVEL,
                           OPTIONS);
   IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR);

!  Switch depending on whether parameter string is a file
!  name, a name pattern, or a DEFINE:

   CASE KIND OF
      BEGIN

   !  If it is a file name:

      0 -> BEGIN

      !  Resolve file name to fully qualified form:

         ERROR := FILENAME_RESOLVE_(
                           CI^STARTUP.PARAM[0]:COUNT,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH);
         IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR)

      !  Call PRINT^NAME to print the file name:

         ELSE CALL PRINT^NAME(FULLNAME,FULL^LENGTH);
      END;
   !  If it is a pattern:

      1 -> BEGIN

      !  Resolve pattern to fully qualified form:

         ERROR := FILENAME_RESOLVE_(
                           CI^STARTUP.PARAM[0]:COUNT,
                           PATTERN:MAXPATTERN,
                           FULL^LENGTH);
         IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR)

      !  Call FIND^FILES to search for and print all file
      !  names that match this pattern:

         ELSE CALL FIND^FILES(PATTERN,FULL^LENGTH);
      END;

   !  If it is a DEFINE:

      2 -> BEGIN

      !  Accept only a map DEFINE, reject all others with
      !  error:

         OPTIONS := DEFINE^SIMPLE^RESOLVE LOR DEFINE^REJECT;

      !  Resolve to fully qualified form of file named by
      !  map DEFINE:

         ERROR := FILENAME_RESOLVE_(
                           CI^STARTUP.PARAM[0]:COUNT,
                           FULLNAME:MAXLEN,
                           FULL^LENGTH,
                           OPTIONS);
         IF ERROR <> 0 THEN CALL FILE^ERRORS(ERROR)

      !  Call PRINT^NAME to print the file name:

         ELSE CALL PRINT^NAME(FULLNAME,FULL^LENGTH);
      END;
   END;
END;
```

14 Using the IOEdit Procedures

The IOEdit package is made up of a set of routines that allow an application to read and write EDIT files (files with file code 101). The procedure call interface to the IO-Edit routines allows access to EDIT files from any supported language: TAL, C, COBOL85, Pascal, and FORTRAN.

This section describes how to use the procedure-call interface to the IOEdit routines. Specifically, it describes how to perform the following operations:

- Create, open, and initialize IOEdit files (OPENEDIT_ procedure)
- Read and write EDIT files (READEDIT[P] and WRITEEDIT[P] procedures)
- Pack and unpack text in an EDIT file line (PACKEDIT and UNPACKEDIT procedures)
- Delete lines from an EDIT file (DELETEEDIT procedure)
- Renumber lines in an EDIT file (NUMBEREDIT procedure)
- Get and set the current-record pointer (GETPOSITIONEDIT and POSITIONEDIT procedures)
- Handle "file full" errors (EXTENDEDIT procedure). The maximum edit file size is 128MB.
- Get and set the record number increment (GETINCREMENTEDIT and INCREMENTEDIT procedures)
- Complete an IOEdit operation in an application that uses nowait I/O (COMPLETEIOEDIT procedure)
- Compress an IOEdit file (COMPRESSEDIT procedure)
- Close an IOEdit file (CLOSEEDIT_ and CLOSEALLEDIT procedures)

When to Use and When Not to Use EDIT Files

Files in EDIT format are intended for text editing applications.

EDIT files are unsuitable for several kinds of applications. Specifically, you must observe the following restrictions on the use of EDIT files:

- You cannot use EDIT files for database purposes.
- There is no locking available for EDIT files.
- NonStop TM/MP protection is not supported for EDIT files.
- You cannot use alternate keys with EDIT files.
- Checkpointing is not supported if your application uses IOEdit procedures.

If your application cannot satisfy all the above restrictions you must use Enscribe files or NonStop SQL files.

Overview of IOEdit

Before discussing how you can use IOEdit to perform operations on EDIT files, some of the major concepts of the IOEdit package will be introduced. This subsection discusses:

- The types of files IOEdit can access
- How to choose between the various tools available for accessing EDIT files
- How lines are typically numbered in an EDIT file and how those line numbers correlate to record numbers
- The purpose of the EDIT file segment (EFS)

When Should You Use IOEdit?

You can access EDIT files using any of the following sets of procedures:

- EDITREADINIT and EDITREAD procedures
- SIO procedures
- IOEdit procedures

EDITREADINIT and EDITREAD work for applications that need only to read EDIT files sequentially. To do more than sequential reads, you should use SIO or IOEdit.

Use Tables 14-1 and 14-2 to establish whether to use IOEdit or SIO in your application to access EDIT files.

Table 14-1 lists the advantages of IOEdit over SIO.

Table 14-1. Advantages of IOEdit Over SIO (page 1 of 2)

If you want to...	Then IOEdit is better than SIO because...
Write a text editor.	IOEdit allows files to be open for input and output at the same time; SIO does not. IOEdit supports deleting records, inserting records, replacing records, backspacing over records, and renumbering records in all or part of a file; SIO supports none of these features.
Perform random access as well as sequential access.	SIO supports only sequential access.
Use an extended data segment for reference parameters.	IOEdit can use extended data segments; SIO cannot.

Table 14-1. Advantages of IOEdit Over SIO (page 2 of 2)

If you want to...	Then IOEdit is better than SIO because...
Sequentially read a file from its start to its middle, then start writing, deleting any existing records from that point; that is, simulate a tape.	You can do this using IOEdit, but not using SIO.
Have faster throughput.	IOEdit is faster because it does not need to perform checksum operations; SIO does need to perform periodic checksum operations to guard against the user program inadvertently overwriting the SIO buffers in the user data segment. IOEdit buffers are protected in the EFS.
Read and write lines in a packed form; for example, when writing a compiler to copy input lines to a scratch file in packed form, and to speed up the scanning of source lines by not having to skip over long strings of blanks one character at a time.	IOEdit supports reading and writing of text lines in packed as well as unpacked form; SIO supports only unpacked records.
Avoid having to use space in the user data segment for input and output buffers.	IOEdit allocates all the space it needs in its own program file segment; SIO uses at least 144 bytes of the user data stack for reading and at least 1024 bytes for writing. For acceptable performance, SIO often needs much larger buffers.
Save disk space on smaller files.	IOEdit assumes default extent sizes of one page for both primary and secondary extents; SIO assumes default sizes of 4 and 16 pages.
Specify sync depth when opening a file.	IOEdit permits you to specify the sync depth; SIO does not.
Have damaged files repaired when opening (like text editors do).	IOEdit can repair damaged EDIT files; SIO cannot.
Avoid having to set up special control blocks for the files you are accessing.	IOEdit procedures need only the file number returned by the FILE_OPEN_ procedure; SIO requires that you set up a separate file control block for each file and a common file block.

Table 14-2 indicates when SIO works better than IOEdit.

Table 14-2. Advantages of SIO Over IOEdit

If you want to...	Then SIO is better than IOEdit because...
Use an error file for reporting errors.	SIO uses an error file; IOEdit does not.
Write long lines.	SIO has a write-fold feature that allows a long line to be divided into several shorter lines; IOEdit has no such feature.
Perform I/O with file types other than EDIT files and T-Text files.	SIO is able to access many device and file types; IOEdit can access only EDIT and T-Text files.

See Section 15, Using the Sequential Input/Output Procedures, for details of using SIO.

Line Numbers and Records

A line number identifies each line in an EDIT file. You can see the line numbers in an EDIT file using the SHOWNUMBER EDIT command, for example:

```
1     ?INSPECT,SYMBOLS
1.001
1.1   ?NOLIST
2     ?SOURCE $SYSTEM.SYSTEM.EXTDECS0(WRITE,READ,FILE_OPEN_,
2.1                       PROCESS_GETINFO_,WRITEREAD,NUMOUT)
3     ?LIST
4
5     PROC MYPROC MAIN;
6     BEGIN
6.003 STRING .TERM^NAME[0:ZSYS^VAL^LEN^FILENAME-1];
6.004 INT NAME^LEN;
6.01  INT TERM^NUM;
6.02  INT .BUFFER[0:127];
6.03  INT WCOUNT;
6.04  STRING .SBUFFER := @BUFFER '<<' 1;
6.05  INT .S^PTR;
6.06  INT ASCII[0:5];
6.07  INT NUM;
6.1
7   CALL PROCESS_GETINFO_(,,,,,TERM^NAME:ZSYS^VAL^LEN^FILENAME-1
                          ,NAME^LEN);
8     CALL FILE_OPEN_(TERM^NAME:NAME^LEN,TERM^NUM);
       .
       .
```

Note that blank lines have a number. Inserted lines use up to three levels of index or “point” numbers. An example of how point numbers might be applied is given below:

If a line is inserted between...	Then the new line is given line number...
lines 6 and 7	6.1
lines 6 and 6.1	6.01
lines 6 and 6.01	6.001
lines 6 and 6.001	6.001, and the original line 6.001 is renumbered

A record number is a line number multiplied by 1000. Line 1 in the previous example is therefore contained in record 1000, line 1.1 in record 1100, and line 6.003 in record 6003. The relationship between the record number and the record itself is maintained in a directory in the file. The record number therefore has a function similar to the key value in an Enscribe key-sequenced file.

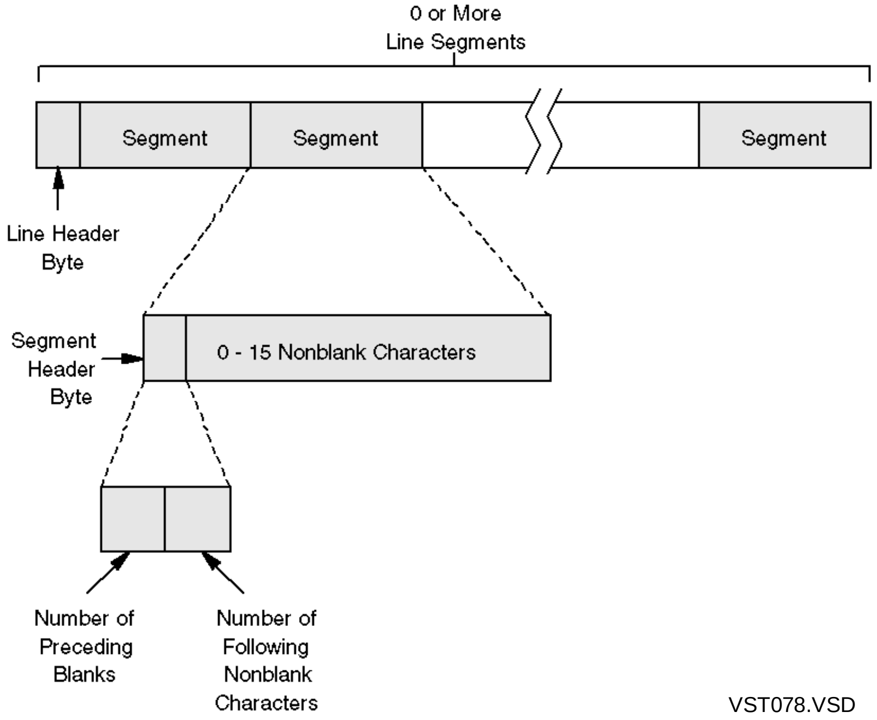
Packed Line Format

Lines are not saved character by character in an EDIT file as in an Enscribe file. Instead, spaces within the line are compressed not only to take up less space on disk but also to reduce the time needed to perform data transfers.

Most programs using IOEdit read and write text-line images in the normal unpacked format; IOEdit converts each line image between the two formats. For efficiency reasons, however, some programs read and write lines directly in the EDIT packed format.

Figure 14-1 shows the format of a packed IOEdit line.

Figure 14-1. An IOEdit Packed Line


0 or More
Line Segments
Segment
Segment
Segment
Line Header
Byte
Segment
Header
Byte
0 - 15 Nonblank Characters
Number of
Preceding
Blanks
Number of
Following
Nonblank
Characters
VST078.VSD

The first byte of the packed line is the header byte. It contains the length of the packed line. If the line is blank, then the header byte contains 0 and there are no following line segments.

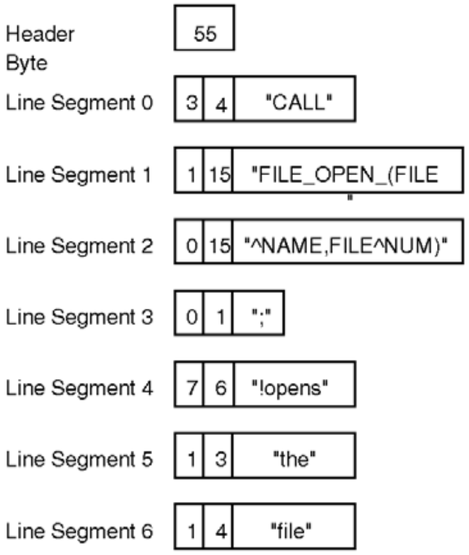
If the header byte is nonzero, then one or more line segments follow. Each line segment contains a header byte and from 0 through 15 nonblank characters. The upper four bits of the header byte indicate the number of blank characters that precede the nonblank characters in the unpacked line. The lower four bits represent the number of nonblank characters in the line segment. A line segment can thus represent up to 15 consecutive blank characters that can precede up to 15 consecutive nonblank characters.

To represent more than 15 consecutive blanks, the packed line uses a single-byte line segment that indicates 15 preceding blanks and no nonblank characters, followed by another line segment indicating some more blanks before the nonblank characters begin. Similarly, more than 15 consecutive nonblank characters are represented by a line segment that indicates some preceding blanks and 15 nonblank characters, followed by a line segment with no preceding blanks and some more nonblank characters.

As an example, consider the following text line from a program source file (the pound signs, #, represent blanks):

```
###CALL#FILE_OPEN_(FILE^NAME,FILE^NUM);#######!opens#the#file
```

The corresponding packed form contains seven line segments and a line header byte as follows:



Header Byte	55		
Line Segment 0	3	4	"CALL"
Line Segment 1	1	15	"FILE_OPEN_(FILE
Line Segment 2	0	15	"^NAME,FILE^NUM)"
Line Segment 3	0	1	";"
Line Segment 4	7	6	"!opens"
Line Segment 5	1	3	"the"
Line Segment 6	1	4	"file"

VST079.VSD

The EDIT File Segment

IOEdit uses a set of data structures in its own extended data segment known as the EDIT file segment (EFS), which is analogous to the PFS used by the file system. This storage area contains buffers that allow the IOEdit routines to read complete pages of text from disk, perform read-ahead operations, and buffer output to the text file. In addition to improving performance, this method also removes the need to use space in the user data segment.

The EDIT file segment is automatically created when you issue an INITIALIZEEDIT procedure call. This call is implied when you open an EDIT file with the OPENEDIT_ procedure. On opening your first EDIT file, you therefore automatically create the EFS. On subsequent opens, you initialize data structures within the EFS for the new file.

IOEdit and Errors

When testing for errors following calls to IOEdit procedures, you need test only for positive errors. IOEdit repairs the file following any negative errors. If you want to handle damaged files directly, for example by sending a message to the user, you can test for a negative error.

Creating, Opening, and Initializing an IOEdit File

Creating, opening, and initializing an EDIT file can all be done with one call to the OPENEDIT_ procedure. Table 14-3 shows when OPENEDIT_ performs each of these functions.

Table 14-3. Functions of the OPENEDIT_ Procedure

If the EDIT file...	Then the OPENEDIT_ procedure call...
Does not yet exist	Creates the file Opens the file Initializes data structures in the EFS for the file
Already exists but is not yet open	Opens the file Initializes data structures in the EFS for the file
Is already open by a call to FILE_OPEN_	Initializes data structures in the EFS for the file

To create a file, the OPENEDIT_ procedure calls the FILE_CREATE_ procedure.

To open a file, the OPENEDIT_ procedure calls the FILE_OPEN_ procedure.

To initialize data structures in the EFS for the EDIT file, the OPENEDIT_ procedure calls the INITIALIZEEDIT procedure. If the EFS does not yet exist, INITIALIZEEDIT creates and allocates the EFS, and then initializes IOEdit data structures within the EFS.

If the process already has the file open using another OPENEDIT_ or OPENEDIT procedure call, then IOEdit flushes the buffers, including directory information, to disk. Hence the file is always current when you open it. If, however, other processes have the file open, then the file state could be inconsistent.

Opening an Already Existing File

If the EDIT file already exists, for example through use of the TEDIT program, then the OPENEDIT_ procedure:

1. Opens the file by calling the FILE_OPEN_ procedure
2. Initializes IOEdit structures on behalf of the file in the EFS by calling the INITIALIZEEDIT procedure

When opening a file with the OPENEDIT_ procedure, you must supply the name of the file you want to open and a negative file number. The negative file number indicates to IOEdit that the file is not yet open. When OPENEDIT_ finishes, the file number of the open file is returned in the `file-number` parameter.

Other parameters for setting the access mode, exclusion mode, nowait mode, and sync depth are all optional. Note that the default values and effects of some of these parameters differ from the corresponding parameters in the FILE_OPEN_ call.

2 The access mode is read-only by default. Note that this is different from FILE_OPEN_ which has a default access mode of read/write.

- The exclusion mode is shared by default, as for the FILE_OPEN_ procedure.
- Nowait I/O is implied by default. Note that this is different from the default mode used by FILE_OPEN_. Moreover, the behavior of nowait I/O is different for IOEdit than for Enscribe files.

  IOEdit passes the nowait attribute to the FILE_OPEN_ procedure to open the EDIT file for nowait I/O. Once the file is open, IOEdit buffers write operations until either a full page of text has been entered or the application calls the CLOSEEDIT_ procedure. IOEdit then writes the entire page to disk with one nowait I/O operation and immediately continues to fill the next page. If you specify waited I/O, then the write to disk is done using waited I/O.

  Whether the file is opened with waited or nowait I/O does not affect the way the application functions: I/O operations return as soon as the access to the buffer in the EFS is complete. Nowait I/O, however, results in improved performance because of the ability to continue writing to the buffers while the last completed page is being written to disk.

- The *`writethrough`* parameter causes write operations to go straight to disk, without using the EFS buffers. Normally, this feature is turned off. Use it with care because the writethrough feature reduces application performance.

The following example opens a text file called DIARY, accepting the default values of shared, read-only access, and nowait I/O for writing out the buffers:

```
FILE^NAME ':=' "$USERVOL.MYSUBVOL.DIARY" -> @S^PTR;
NAME^LEN := @S^PTR '-' @FILE^NAME;
FILE^NUM := -1;
ERROR := OPENEDIT_(FILENAME:NAME^LEN,
                   FILE^NUM,
                   !access^mode!,
                   !exclusion^mode!,
                   !nowait^mode!,
                   !sync^depth!,
                   !writethrough!);
IF ERROR > 0 THEN ...
```

Opening a Nonexistent File

To open an EDIT file that does not yet exist, the OPENEDIT_ procedure:

1. Creates the EDIT file by calling the FILE_CREATE_ procedure
2. Opens the file by calling the FILE_OPEN_ procedure
3. Initializes the IOEdit data structures in the EFS by calling the INITIALIZEEDIT procedure

To create a new file, you must set the access mode to read/write. Otherwise, there is no difference between calling OPENEDIT_ to open an existing file and calling OPENEDIT_ to open a file that does not yet exist. The following example creates and opens a file called MYFILE:

```
LITERAL READ^WRITE = 0;
  .
  .

FILE^NAME ':=' "$USERVOL.MYSUBVOL.MYFILE" -> @S^PTR;
NAME^LEN := @S^PTR '-' @FILE^NAME;
FILE^NUM := -1;
ACCESS^MODE := READ^WRITE;
ERROR := OPENEDIT_(FILENAME:NAME^LEN,
                   FILE^NUM,
                   ACCESS^MODE);
IF ERROR > 0 THEN ...
```

The OPENEDIT_ procedure returns file-system error 11 if the file is not open for read/write access.

Initializing an Already Open File

If an EDIT file has already been opened using, for example, the FILE_OPEN_ procedure, then you can initialize the IOEdit data structures in the EFS by calling the OPENEDIT_ procedure. The OPENEDIT_ procedure establishes that the file exists and that it is an EDIT file and then calls the INITIALIZEEDIT procedure to perform the initialization.

To initialize an already open file, you must supply OPENEDIT_ with the file number returned by the FILE_OPEN_ procedure; for example:

```
FILE^NAME ':=' "$USERVOL.MYSUBVOL.DIARY" -> @S^PTR;
NAME^LEN := @S^PTR '-' @FILE^NAME;
ERROR := FILE_OPEN_(FILE^NAME:NAME^LEN,FILE^NUM);
IF ERROR <> 0 THEN  ...
  .
  .
ERROR := OPENEDIT_(FILENAME:NAME^LEN,
                   FILE^NUM);
IF ERROR > 0 THEN ...
```

Reading and Writing an IOEdit File

This subsection discusses some of the operations you can perform that relate to I/O with EDIT files. Specifically, it covers how to perform the following operations:

- Set the starting point for a sequential I/O operation using the POSITIONEDIT, READEDIT, or WRITEEDIT procedure.
- Perform sequential reading.
- Perform sequential writing, including how to append lines to the end of a file, how to insert lines into a file, and how to handle error 45, the File Full error.
- Delete lines from a file.
- Renumber lines in a file.
- Set and get the record number increment.
- Perform line backspacing, as required to support the FORTRAN BACKSPACE statement.

Record Pointers

Like the Enscribe database record manager, IOEdit makes use of current-record and next-record pointers when performing I/O with EDIT files.

The current-record pointer points to the record that the last I/O operation accessed; that is, the last record read, the last record written to, or the record that precedes a set of deleted records.

The next-record pointer points to the record that the next operation will be performed on. For a read operation, the next record is the record in the file that immediately follows the current record. For a write operation, the next record number is the current record number plus the current record number increment. The record number increment is explained later in this subsection.

In addition to record numbers that relate to line numbers as described earlier in this section, there are some special values for a record number, as shown in Table 14-4:

Table 14-4. Record Numbers

Number	Explanation
-1	The lowest-numbered line in the file
-2	The highest-numbered line in the file
-3 or unspecified	The current record as indicated by the last I/O operation
0 or greater	The desired line number times 1000

Selecting a Starting Point

Most I/O operations involve reading or writing sequential data. However, unlike other available methods of access to EDIT files, the IOEdit routines allow you to start your read or write operation at any record in the file. This record can be:

- The beginning of the file (record number -1)
- The end of the file (record number -2)
- Any specified record number
- The record indicated by the next-record pointer (record number -3)

You can set the starting position for a series of one or more sequential I/O operations using one of the following operations:

- Specify the record number in a call to the POSITIONEDIT procedure.
- Specify the record number directly in the call to READEDIT.
- Specify the record number directly in a call to WRITEEDIT.

Using the POSITIONEDIT Procedure

The following example sets the value of the next-record pointer to line 50 using the POSITIONEDIT procedure:

```
RECORD^NUMBER := 50000D;
ERROR := POSITIONEDIT(FILE^NUM,RECORD^NUMBER);
IF ERROR > 0 THEN ...
```

Using the READEDIT Procedure

The following example sets the value of the next-record pointer to the first line in the file by supplying -1D as the record number to the READEDIT procedure:

```
RECORD^NUMBER := -1D;
ERROR := READEDIT(FILE^NUM,RECORD^NUMBER,BUFFER,BUFFER^LEN,
                  BYTES^READ);
IF ERROR > 0 THEN ...
```

If you supply the READEDIT procedure with a record number that does not exist, then the next record is assumed to be the next-higher record number that does exist.

Using the WRITEEDIT Procedure

The following example sets the value of the next-record pointer to the end of the file using the WRITEEDIT procedure:

```
RECORD^NUMBER := -2D;
ERROR := WRITEEDIT(FILE^NUM,RECORD^NUMBER,BUFFER,WCOUNT);
IF ERROR > 0 THEN ...
```

The record number you supply to WRITEEDIT must not already exist; otherwise, error 10 is returned.

Performing Sequential Reading

Once you have established the starting point for a sequential read operation, you can read each record in turn by repeated calls to either the READEDIT or READEDITP procedure. READEDIT puts the text line in the application buffer in the unpacked state; that is, IOEdit unpacks the record from the form in which it is stored on disk.

READEDITP returns packed text into the application buffer. You can unpack packed text using the UNPACKEDIT procedure, if desired.

To read sequentially from an EDIT file, each successive READEDIT (or READEDITP) operation must not specify the record number. By default, IOEdit reads the record that is logically next in the file.

The following example sequentially reads an entire file:

```
LITERAL START^OF^FILE = -1;
 .
 .
RECORD^NUMBER := START^OF^FILE;
ERROR := POSITIONEDIT(FILE^NUM,RECORD^NUMBER);
IF ERROR > 0 THEN ...

WHILE ERROR > 1 DO
BEGIN
   ERROR := READEDIT(FILE^NUM,
                    !record^number!,  !unspecified
                    BUFFER,           !contains text read
                    BUFFER^LEN,       !length of input buffer
                    BYTES^READ);      !number of bytes read
   IF ERROR > 0 AND ERROR <> 1 THEN ...
    .
    .
END;
```

Performing Sequential Writing

Before you can write to a file, you must first open it for either write-only or read/write access. By default, OPENEDIT_ opens a file with read-only access (unlike FILE_OPEN_, which defaults to read/write). If you do open a file for write-only access, IOEdit actually opens it for read/write access because IOEdit needs to read the file directory.

Once you have established the start point for your sequential write operation, you can issue repeated calls to WRITEEDIT or WRITEEDITP. WRITEEDIT causes IOEdit to pack the line of text you are writing before copying it to the disk. WRITEEDITP assumes that the line of text is already packed, for example by using the PACKEDIT procedure.

Typically, when you write lines to a file, you either append them to the end of the file or insert them between existing lines in the file. The following paragraphs describe how to perform these operations.

Appending to a File

To append new lines of text to the end of an EDIT file you must:

1. Position the next-record pointer to the end of the file.
2. Set the record increment to the desired value.
3. Issue repeated write operations, one for each line of text.

The record increment you set in Step 2 determines the difference in record numbers between logically adjacent records as you add them to the file. For example, if the last line in the file is line 60 (record 60000) and the record increment is 1000, then successive records will have record numbers 61000, 62000, and so on.

The following example writes unpacked lines to the end of an EDIT file until the user presses the F1 key. This example uses a record increment of 1000:

```
LITERAL LAST^RECORD = -2D,
        F1 = ...;

RECORD^NUMBER := LAST^RECORD;
ERROR := POSITIONEDIT(FILE^NUM,RECORD^NUMBER);
IF ERROR > 0 THEN ...

!Set the record number increment:
DELTA := 1000D;
CALL INCREMENTEDIT(FILE^NUM,DELTA);

!Read text line from terminal:
BUFFER ':=' "> ";
WCOUNT := 2;
RCOUNT := 80;
CALL WRITEREAD(TERM^NUM,BUFFER,WCOUNT,RCOUNT,BYTES^READ);

!If not function key 1, write to end of file and read next
!line.  Repeat until user presses F1:
WHILE FIRST^BYTE <> F1 DO        !FIRST^BYTE is a bytes
BEGIN                            ! pointer to BUFFER
   ERROR := WRITEEDIT(FILE^NUM,
                      !record^number!,
                      BUFFER,WCOUNT);
   BUFFER ':=' "> ";
   WCOUNT := 2;
   CALL WRITEREAD(TERM^NUM,BUFFER,RCOUNT,BYTES^READ);
END;
```

Inserting Lines

Typically, you insert text after the current line in the file. To do this, you need to choose an ascending sequence of record numbers that are all greater than the current record number and all less than the next record number. The following sequence outlines one approach:

1. Determine the current record number using the GETPOSITIONEDIT procedure.
2. Determine the appropriate record number increment to use by checking the increment between the current-record and next-record pointers.

   For example, if the current position is record 5000 and the next record is record 5100, then a record increment of 10 is appropriate. Inserted lines then have record numbers 5010, 5020, and so on.

   Use the INCREMENTEDIT procedure to set the increment.
3. Start writing records.
4. If any write operation returns error 10, then you are trying to overwrite an existing record. Return to Step 2 and use a smaller increment.
5. If a smaller increment is not available, you must renumber the subsequent text line(s) to make room in the record-numbering scheme for the additional lines.

Setting and Getting the Record Number Increment

When you write or renumber records in an EDIT file, each record number differs from the previous record number by the record number increment.

The INCREMENTEDIT and GETINCREMENTEDIT procedures allow you to control the record number increment. The following example gets the current record number increment:

```
INT(32) INCREMENT;
.
.
INCREMENT := GETINCREMENTEDIT(FILE^NUM);
```

The next example sets the record number increment to 100:

```
DELTA := 100D;
CALL INCREMENTEDIT(FILE^NUM,DELTA);
```

Renumbering Lines

When inserting lines into an EDIT file, you sometimes need to renumber some lines because you have exhausted the possible record numbers in the range between the record preceding the inserted text and the record that follows the inserted text. For example, if you try to insert text between records 1001 and 1002, then record 1002 will have to be renumbered.

To renumber lines, supply the NUMBEREDIT procedure with the range of lines you need to have renumbered, the new record number for the start of the renumbered set of lines, and the record number increment.

The following example renumbers records 51200 through 80000, starting the new record number range at 60000 and with a record number increment of 10:

```
FIRST := 51200D;
LAST := 80000D;
START := 60000D;
INCREMENT := 10;
ERROR := NUMBEREDIT(FILE^NUM,FIRST,LAST,START,INCREMENT);
IF ERROR > 0 THEN ...
```

Handling “File Full” Errors

Error 45 (file full) can sometimes return from a write operation, indicating that all the space allocated to the file has been used. You can use the EXTENDEDIT procedure to increase the extent size of the file; for example:

```
ERROR := EXTENDEDIT(FILE^NUM);
```

EXTENDEDIT functions as follows:

1. Creates a new file with the extended extent size
2. Copies the contents of the old file into the new file
3. Deletes the old file
4. Names the new file with the same name as the old file
5. Returns the new file number in the *`file-number`* parameter

The new file retains the same line numbering as the old file unless you specify a starting record number and record increment in the call to EXTENDEDIT. The following example renumbers the lines in the file, starting at line number 1, with a record number increment of 1000:

```
START := 1000D;
INCREMENT := 1000D;
ERROR := EXTENDEDIT(FILE^NUM,START,INCREMENT);
```

Deleting Lines

Another common operation performed on EDIT files is to delete lines of text. To do this, you supply the DELETEEDIT procedure with the range of lines that you want to delete. DELETEEDIT deletes all lines with record numbers greater than or equal to the specified starting record and less than the specified last record.

The following example deletes the lines between two records obtained by calls to the GETPOSITIONEDIT procedure:

```
START^DELETE := GETPOSITIONEDIT(FILE^NUM);
 .
 .
END^DELETE := GETPOSITIONEDIT(FILE^NUM):

ERROR := DELETEEDIT(FILE^NUM,START^DELETE,END^DELETE);
IF ERROR > 0 THEN ...
```

Line Backspacing

The BACKSPACEEDIT procedure performs the equivalent of a FORTRAN BACKSPACE statement on the file.

The purpose of the BACKSPACEEDIT procedure is as follows. The ANSI FORTRAN standard specifies that the ENDFILE statement writes an end-of-file record and that a BACKSPACE statement backspaces over that record if it follows an ENDFILE statement or a READ statement that returned the end-of-file status. The EDIT file format does not provide an end-of-file marker; the logical end-of-file is immediately after the highest-numbered line in the file.

To satisfy the ANSI requirement, the FORTRAN ENDFILE statement and READ end-of-file will both set the file’s current record number to -2, indicating that a simulated end-of-file record has just been read or written; in this case, the BACKSPACEEDIT procedure clears the end-of-file status indication by setting the current record number to that of the highest-numbered line in the file. Table 14-5 summarizes the effect of the BACKSPACEEDIT procedure:

Table 14-5. Effects of the BACKSPACEEDIT Procedure

If the current record number is...	Then the BACKSPACEEDIT Procedure...
-1	Does nothing
-2	Sets the current record number to the highest-numbered record in the file, or to -1 if the file is empty
Greater than or equal to 0	Sets the current record number to the number of the preceding record in the file, or to -1 if no such record exists

The procedure call takes the EDIT file number as its only parameter and returns a file-system error code:

```
ERROR := BACKSPACEEDIT(FILE^NUM);
IF ERROR > 0 THEN ...
```

Using Nowait I/O With IOEdit Files

If your program issues a call to AWAITIO with -1 specified for the file number (meaning wait until any outstanding I/O request from this process is finished), you may get a completion status for a nowait I/O operation previously started by IOEdit. If such an AWAITIO call returns a file number that could be that of an IOEdit file, then your program must use the COMPLETEIOEDIT procedure to inform IOEdit that the I/O request is complete and pass to IOEdit the results that AWAITIO returned.

The COMPLETEIOEDIT procedure returns a function value of -1 if the file is being managed by IOEdit; otherwise, it returns 0.

The following example checks the outcome of an AWAITIO call to see if the completing I/O operation is managed by IOEdit or the file system:

```
CALL AWAITIO(FILE^NUM,BUFFER,BYTES^READ,TAG);
IF COMPLETEIOEDIT(FILE^NUM,BUFFER,BYTES^READ,TAG)
   THEN                      !IOEdit file
   ELSE                      !other file
```

See Section 4, Using Nowait Input/Output, for a detailed discussion of nowait I/O.

Compressing an IOEdit File

Compressing files is done to save disk space. Because of the way IOEdit manages its files, areas of unused space may occur within the space allocated to the file.

You can compress an EDIT file using the COMPRESSEDIT procedure. This procedure copies a file line by line, making each block as full as possible, thereby minimizing the number of disk pages occupied by the file. You can do the same thing using the PUT! command of the EDIT program.

On completion, COMPRESSEDIT returns the file number of the compressed file in the *`file-number`* parameter. The old file gets deleted. The current-record pointer identifies the first line in the new file (current record number -1).

You identify the file you want compressed by supplying the *`file-number`* parameter. Optionally, you can also specify the record number of the first line of the new file and the record number increment. If you do not supply these optional parameters, the line numbering remains unchanged.

The following example compresses a file and renumbers the lines, starting at line 1, with a record number increment of 1000:

```
START := 1000D;
INCREMENT := 1000D;
ERROR := COMPRESSEDIT(FILE^NUM,START,INCREMENT);
IF ERROR > 0 THEN ...
```

Closing an IOEdit File

This subsection discusses how to use the CLOSEEDIT_ and CLOSEALLEDIT procedures. In addition to closing EDIT files, these procedures also cause the IOEdit buffers in the EFS to be copied to disk.

△ **Caution.** You must close EDIT files explicitly. If you allow EDIT files to be closed implicitly by stopping the process, then you will lose the contents of the IOEdit buffers.

Closing a Single File

You usually close EDIT files one at a time using the CLOSEEDIT_ procedure. You supply the procedure with the file number; for example:

```
ERROR := CLOSEEDIT_(FILE^NUM);
IF ERROR > 0 THEN ...
```

IOEdit responds by closing the file and copying the buffers to disk.

Closing All EDIT Files

You can close all EDIT files that your process has open by issuing a call to the CLOSEALLEDIT procedure. You might do this, for example, in a signal handler or trap handler to save the file buffers before stopping the process. For example:

```
CALL CLOSEALLEDIT;
```

See Section 25, Debugging, Trap Handling, and Signal Handling, for a discussion of trap handlers.

15

Using the Sequential Input/Output Procedures

The sequential input/output (SIO) procedures provide a higher-level interface than the interface provided by using the file system procedures directly. They are intended for processing sequential I/O streams, particularly of displayable or printable text. Specifically, they can be used for text that might be directed to or from a variety of text sources or destinations, such as terminals, printers, spoolers, structured disk files, and EDIT files. Files opened for SIO access are referred to as SIO files.

SIO procedures provide a convenient way of reading or writing EDIT files as text files, ignoring such things as line numbers. EDIT files have a higher-level structure that the file system does not understand but which SIO does understand. An alternative to using SIO would be to use the IOEdit procedures as described in Section 14, Using the IOEdit Procedures.

SIO is designed to work with the INITIALIZER, which allows redirection of SIO files using the Startup message and Assigns.

This section shows how to use the SIO procedures in an application program. It explains how to program the following operations:

- Initialize file control blocks (FCBs) for SIO files using TAL or pTAL DEFINEs.
- Open SIO files using the OPEN^FILE procedure.
- Retrieve information (such as the current file state and permissions) about SIO files using the CHECK^FILE procedure.
- Read and write to an SIO file using the READ^FILE and WRITE^FILE procedures.
- Access files in EDIT format.
- Handle nowait input and output.
- Communicate with other processes.
- Handle system messages using SIO procedures.
- Handle the BREAK key.
- Handle errors that occur in response to an SIO procedure call.
- Close SIO files.
- Dynamically initialize FCBs for SIO files without using TAL or pTAL DEFINEs.

An Introduction to the SIO Procedures

An application process can use the following SIO procedures to sequentially access files:

CHECK^BREAK	Checks whether the BREAK key has been pressed.
CHECK^FILE	Retrieves characteristics and state information about SIO files.
CLOSE^FILE	Closes a file that was opened for SIO.
GIVE^BREAK	Disables BREAK processing by returning BREAK ownership to the process that this process took the ownership from.
NO^ERROR	Allows SIO processing of errors from non-SIO operations.
OPEN^FILE	Opens a file for access by other SIO procedures. This procedure can also assign file-transfer characteristics.
READ^FILE	Reads a record into a read buffer from a file opened for SIO.
SET^FILE	Sets or changes the characteristics of files accessed by the SIO procedures. These characteristics include modes of access and exclusion, file-transfer attributes, and mode of error processing.
TAKE^BREAK	Enables BREAK processing by the process that issues the TAKE^BREAK call. This call also disables BREAK processing by the current BREAK owner.
WAIT^FILE	Waits for the completion of an outstanding I/O operation initiated on a file opened for nowait SIO.
WRITE^FILE	Writes a record from a write buffer into a file opened for SIO.

Refer to the *Guardian Procedure Calls Reference Manual* for complete details on the procedures listed above.

Refer to the *Guardian Procedure Errors and Messages Manual* for details on specific SIO errors. The SIO procedures may return these messages in addition to regular file-system error messages.

FCBs for SIO Files

The SIO procedures access each file using a special file control block (FCB) in the user’s data area. This FCB contains file information in addition to the information contained in the FCB automatically created and managed by the file system. Each SIO FCB must be programmatically created as described later in this section.

In addition to an FCB for each file, you also need a common FCB for the process. The common FCB contains information common to all SIO files opened by the process. This information includes the address of the FCB that receives error messages generated by the SIO procedures.

For ease of programming, all structures and literals required by the SIO procedures, including the FCB and the common FCB, are predefined in a file called GPLDEFS in the $SYSTEM.SYSTEM subvolume. This file must be sourced into your program if you intend to use the SIO procedures.

Steps for Writing a Program

To use the SIO procedures, your program must perform the sequence of operations outlined below:

1. Initialize every FCB that your process will use. You must initialize one FCB for each file to be accessed by SIO procedures, and a common FCB. You can do this in one of the following ways:
    - Using TAL or pTAL DEFINEs, you can allocate FCBs with some values already initialized. Allocation is static and is done at compile time. The INITIALIZER procedure provides further initial values for the FCBs, such as information provided in ASSIGN commands. INITIALIZER also provides a convenient way to complete FCB initialization without having to directly handle the $RECEIVE file. The subsection Initializing SIO Files Using TAL or pTAL DEFINEs, later in this section, provides details.
    - You can allocate space for FCBs and initialize them by issuing SET^FILE procedure calls. This method does not have the convenience of the INITIALIZER procedure, but it gives you the flexibility to dynamically allocate FCBs and is therefore appropriate if you do not know how many FCBs you will need. The $RECEIVE file is handled directly. Initializing SIO Files Using TAL or pTAL DEFINEs, later in this section, provides details.
    - You can mix the above methods: you can allocate some FCBs using TAL or pTAL DEFINEs and dynamically allocate additional FCBs and initialize them using SET^FILE procedure calls.

Note. TNS/R native callers cannot use TNS FCBs, nor can TNS callers use native FCBs.

2. Open each FCB required by the program. You must use the OPEN^FILE procedure as described in the subsection Opening and Creating SIO Files, later in this section.
3. Perform any other SIO operations as required by your application. These operations may include reading or writing the SIO files or other operations such as processing the BREAK key.
4. Close the SIO files as described in the subsection Closing SIO Files, later in this section.

Differences Between TNS/R Native and TNS Procedures

Most of the SIO procedures can be called by either a TNS/R native caller or a TNS caller with no changes. However, three of the procedure calls, the SET^FILE, CHECK^FILE, and INITIALIZER calls, have different forms in the native and TNS environments.

SET^FILE and CHECK^FILE Differences

The reason for the differences between the native and TNS forms of the SET^FILE and CHECK^FILE procedure calls has to do with restrictions on the way address parameters are passed and returned: in the TNS environment, address values can be passed through a type INT parameter; however, this is not possible in the native environment.

When non-address parameters are passed or returned by SET^FILE or CHECK^FILE, the same form of the calls can be used in either environment. However, when an address value is passed or returned, an additional parameter is required in native mode. In this case, the native forms of the calls are:

```
CALL CHECK^FILE (FCB, OPERATION, ADDR)
```

```
ERROR := SET^FILE (FCB, OPERATION,,, ADDR)
```

where *addr* is a parameter of type WADDR.

For ease in writing programs to be executed in both the native and TNS environments, two DEFINEs are provided in the $SYSTEM.SYSTEM.GPLDEFS file. These DEFINEs call the correct version of SET^FILE or CHECK^FILE for operations that can return an address, depending on which environment the program is compiled in.

The DEFINE for SET^FILE has the form:

```
CALL_SET^FILE_ADDRESS_ (ERROR, FCB, OPERATION, ADDR)
```

The DEFINE for CHECK^FILE has the form:

```
CALL_CHECK^FILE_ADDRESS_ (ERROR, FCB, OPERATION, ADDR)
```

INITIALIZER Differences

In the TNS environment, the INITIALIZER procedure requires that the RUCP and all FCBs be contiguous. In TNS procedures, the RUCP and FCBs are always contiguous; however, in native procedures, this cannot be guaranteed. Therefore, the requirement has been lifted in the native environment. However, two additional parameters are required when calling INITIALIZER from a native procedure to place information in the FCBs.

For TNS callers, the basic INITIALIZER call has the form:

```
CALL INITIALIZER(CONTROL^BLOCK);
```

For native callers, the basic INITIALIZER procedure call has the form:

```
CALL INITIALIZER(CONTROL^BLOCK,,,,,,,NUM^FCBS,FCB^ARRAY);
```

where NUM^FCBS is an INT parameter specifying the number of FCBs and FCB^ARRAY is an array of type WADDR containing pointers to the FCBs.

For convenience in writing programs to run in both the native and TNS environments, the native form of the INITIALIZER call can also be used in TNS procedures, in which case it overrides the processing of contiguous FCBs.

See Using the INITIALIZER Procedure, later in this section, for more information about calling INITIALIZER.

Initializing SIO Files Using TAL or pTAL DEFINEs

A set of TAL and pTAL DEFINEs in the $SYSTEM.SYSTEM.GPLDEFS file enable you to create and initialize an FCB for each SIO file, a common FCB, and a run-unit control block (RUCB):

ALLOCATE^CBS	Allocates and initializes the RUCB and common FCB. It initializes the RUCB with the number of FCBs to be processed by the INITIALIZER.
ALLOCATE^CBS^D00	Performs the same functions as ALLOCATE^CBS, but allocates a larger common FCB. This DEFINE must be used if any of the FCBs are allocated using ALLOCATE^FCB^D00.
ALLOCATE^FCB	Allocates space for and initializes an FCB with the default file name. This DEFINE is typically used for FCBs other than $RECEIVE and the common FCB.
ALLOCATE^FCB^D00	Performs the same functions as ALLOCATE^FCB, but allocates a larger FCB. This DEFINE must be used for the $RECEIVE file.

Note. The FCBs for a TNS/R native process use more memory than those for a TNS process.

The INITIALIZER handles the Startup and Assign messages and places any relevant information from these messages into the appropriate FCBs if the RUCB is passed:

- The Startup message provides the names of the input and output files typically supplied by the user with the RUN command.
- Assign messages provide the actual file name as well as other file characteristics such as access mode and record and block length. These messages result from ASSIGN operations set up by the user before running the program.

The INITIALIZER procedure automatically reads and processes the Startup and Assign messages. File characteristics provided by the program user through Assign messages during process startup can also be provided programmatically using the SET^FILE procedure.

To perform initialization using TAL or pTAL DEFINEs, your program must do the following:

1. Allocate space for the SIO data structures using TAL or pTAL DEFINEs provided in the $SYSTEM.SYSTEM.GPLDEFS file. You need to allocate space for each FCB, the RUCB, and the common FCB. The DEFINEs also provide initial values.
2. Assign a logical file name to each file that the SIO procedures will access (optional).
3. Complete the initialization of the FCBs by calling the INITIALIZER procedure. INITIALIZER uses information from the Startup and Assign messages to supplement information already in the FCBs.

Note. The TNS/R native form of the INITIALIZER procedure call differs from the TNS form. See "Using the INITIALIZER Procedure," later in this section, for more details.

4. Set file characteristics such as access mode, block size, and extent size for each SIO file (optional). These characteristics can supplement or override those already written to the FCBs by the INITIALIZER procedure.

The following paragraphs describe how to perform these steps.

Setting Up the SIO Data Structures

Setting up the FCBs, the common FCB, and the RUCB for the INITIALIZER procedure requires the use of some TAL or pTAL DEFINEs and literals that are described in the $SYSTEM.SYSTEM.GPLDEFS file. To use these templates, you must source this file into your program with a compiler directive as follows:

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
```

Setting Up the Run-Unit Control Block and the Common File Control Block

Use a DEFINE named ALLOCATE^CBS to set up the RUCB and the common FCB. You must specify the following information:

- The name of the RUCB. You will pass this name to the INITIALIZER procedure.
- The name of the common FCB. You will pass this name to the OPEN^FILE procedure.
- The number of additional FCBs that the INITIALIZER procedure will prepare. This number must be the total number of files that INITIALIZER will access.

The following example sets up an RUCB named CONTROL^BLOCK. It specifies a common FCB named COMMON^FCB. The INITIALIZER procedure will set up two additional FCBs:

```
LITERAL NUMBER^OF^FCBS = 2;
ALLOCATE^CBS(CONTROL^BLOCK,
             COMMON^FCB,
             NUMBER^OF^FCBS);
```

Preparing the SIO File Control Blocks

Use a DEFINE named ALLOCATE^FCB or a DEFINE named ALLOCATE^FCB^D00 to set up an FCB for each file that the SIO procedures will access.

Use ALLOCATE^FCB^D00 for $RECEIVE and the common FCB. The created FCB will identify its opener by process handle.

Use ALLOCATE^FCB for all files other than $RECEIVE and the common FCB.

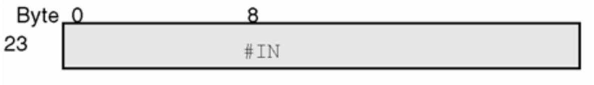
Note. These DEFINE calls must immediately follow the ALLOCATE^CBS^DOO DEFINE call, and they must allocate space for exactly the number of DEFINES specified in the ALLOCATE^CBS^D00 DEFINE call.

You must specify the following information each time you use a DEFINE to allocate an FCB:

- The name of the FCB. This name is used to refer to the file in other SIO procedure calls.
- A physical file name that the FCB will default to.

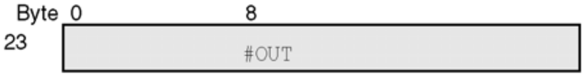
The physical file name is 12 words long. These 12 words can contain either a string that is to be substituted or a complete file name. A string for substitution can be replaced by the input file name from the Startup message, the output file name from the Startup message, the home terminal name, or a temporary file name. The following substitution strings are valid:

- To substitute the input file name from the Startup message, use the following string:



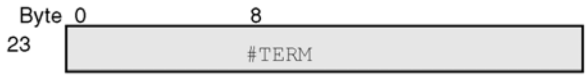
VST065.VSD

- To substitute the output file name from the Startup message, use the following string:



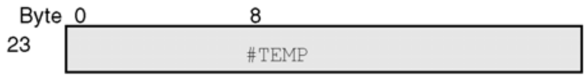
VST066.VSD

- To substitute the home terminal name, use the following string:



VST067.VSD

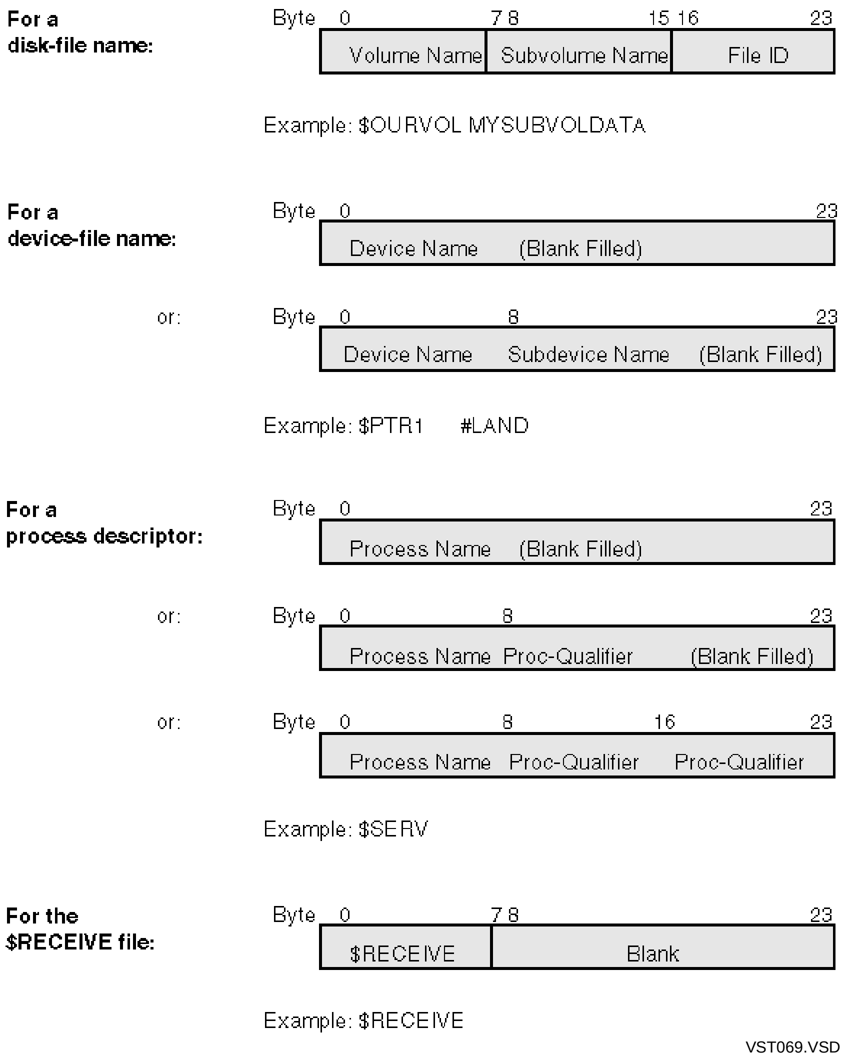
- To substitute a temporary file name, use the following string:



VST068.VSD

To specify a complete file name, you also use exactly 12 words. The format of the name depends on whether the file is a disk file, a process, or a device file. Figure 15-1 shows the valid formats. All fields must be padded with blanks to ensure that the name consists of 24 bytes.

Figure 15-1. File-Name Conventions for SIO File


For a disk-file name:
Byte 0 7 8 15 16 23
Volume Name
Subvolume Name
File ID
Example: $OURVOL MYSUBVOLDATA
For a device-file name:
Byte 0 23
Device Name (Blank Filled)
or:
Byte 0 8 23
Device Name Subdevice Name (Blank Filled)
Example: $PTR1 #LAND
For a process descriptor:
Byte 0 23
Process Name (Blank Filled)
or:
Byte 0 8 23
Process Name Proc-Qualifier (Blank Filled)
or:
Byte 0 8 16 23
Process Name Proc-Qualifier Proc-Qualifier
Example: $SERV
For the $RECEIVE file:
Byte 0 7 8 23
$RECEIVE
Blank
Example: $RECEIVE
VST069.VSD

To access files across the network, you need to include the network number in the file name. Indicate a network number by putting a backslash (\) in the first byte and the system number in the second byte. The dollar sign ($) that normally starts the volume, process, or device name is then omitted, leaving 6 bytes to identify the volume, process, or device. This is one byte less than can be used for local names.

To obtain a system number, you use the LOCATESYSTEM procedure.

The following examples allocate space for two FCBs, one for the input file and one for the output file. The default file names for the input and output files will be read from the Startup message by the INITIALIZER procedure:

```
ALLOCATE^FCB(INFILE,"        #IN             ");
ALLOCATE^FCB(OUTFILE,"        #OUT            ");
```

Note. The TNS/R native form of the INITIALIZER procedure call differs from the TNS form. See Using the INITIALIZER Procedure, later in this section, for more details.

The following example allocates space for an FCB for an explicitly named disk file:

```
ALLOCATE^FCB(DFILE,"$OURVOL MYSUBVOLDATA    ");
```

Assigning a Logical File Name

To enable ASSIGN commands to set file characteristics through the INITIALIZER, you must provide a logical file name for the FCB. This step is redundant if your program will always set file characteristics using only the SET^FILE procedure.

To assign a logical file name, supply the SET^FILE procedure with the address of an array. The first byte of the array indicates the number of characters in the name; subsequent bytes contain the name, which can be up to seven characters long.

The following example provides logical file names for the FCBs allocated above. Three versions of the example are shown: a pTAL version, a TAL version, and a version that uses the CALL_SET^FILE_ADDRESS_ DEFINE to call the correct form of SET^FILE in either environment. See Differences Between TNS/R Native and TNS Procedures, earlier in this section, for details.

pTAL Example:

```
INT .BUF[0:11];
STRING .SBUF = BUF;
 .
 .
SBUF ':=' [5, "INPUT"]
CALL SET^FILE(INFILE,
              ASSIGN^LOGICALFILENAME,
              ,,
              @BUF);

SBUF ':=' [6, "OUTPUT"];
CALL SET^FILE(OUTFILE,
              ASSIGN^LOGICALFILENAME,
              ,,
              @BUF);

SBUF ':=' [5, "LFILE"];
CALL SET^FILE(DFILE,
              ASSIGN^LOGICALFILENAME,
              ,,
              @BUF);
```

TAL Example:

```
INT .BUF[0:11];
STRING .SBUF := @BUF '<<' 1;
 .
 .
SBUF ':=' [5, "INPUT"];
CALL SET^FILE(INFILE,
              ASSIGN^LOGICALFILENAME,
              @BUF);

SBUF ':=' [6, "OUTPUT"];
CALL SET^FILE(OUTFILE,
              ASSIGN^LOGICALFILENAME,
              @BUF);

SBUF ':=' [5, "LFILE"];
CALL SET^FILE(DFILE,
              ASSIGN^LOGICALFILENAME,
              @BUF);
```

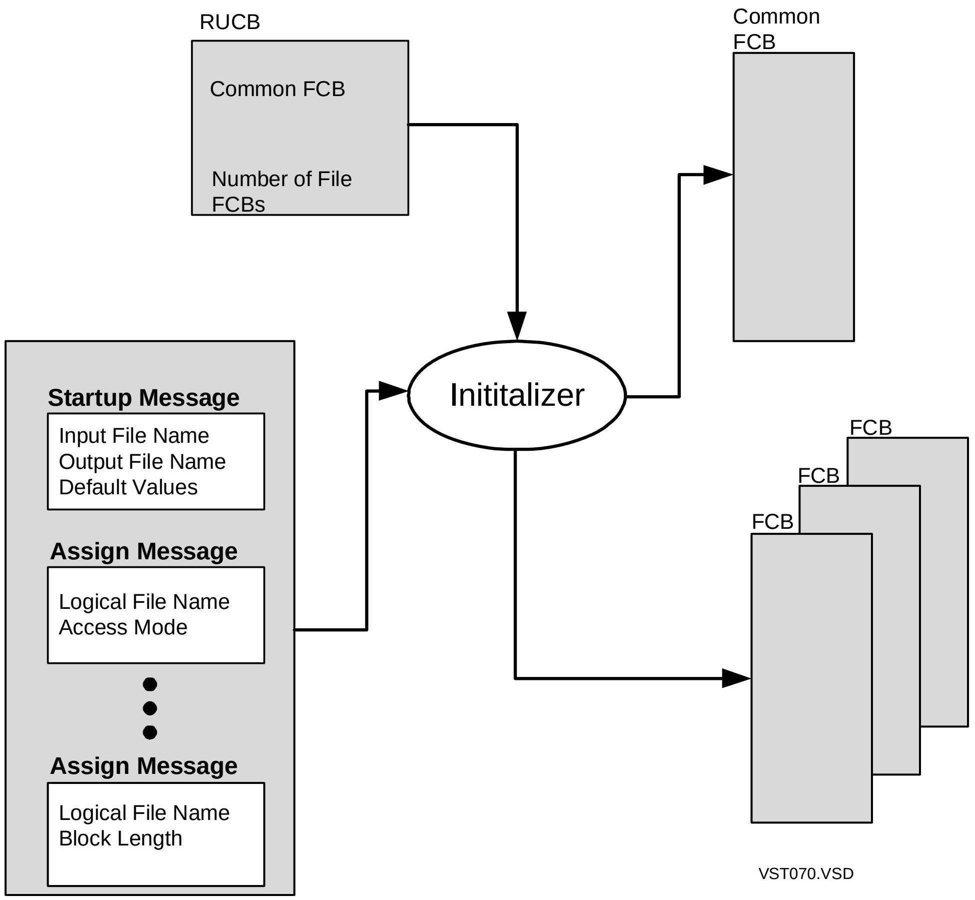
pTAL/TAL Example:

```
INT ERROR;
INT .BUF[0:11];
?IF PTAL     !Begin pTAL statements
STRING .SBUF = BUF;
?ENDIF PTAL  !End pTAL statements
?IFNOT PTAL  !Begin TAL statements
STRING .SBUF := @BUF '<<' 1;
?ENDIF PTAL  !End TAL statements
 .
 .
SBUF ':=' [5, "INPUT"]
CALL_SET^FILE_ADDRESS_(ERROR,
                       INFILE,
                       ASSIGN^LOGICALFILENAME,
                       @BUF);

SBUF ':=' [6, "OUTPUT"];
CALL_SET^FILE_ADDRESS_(ERROR,
                       OUTFILE,
                       ASSIGN^LOGICALFILENAME,
                       @BUF);

SBUF ':=' [5, "LFILE"];
CALL_SET^FILE_ADDRESS_(ERROR,
                       DFILE,
                       ASSIGN^LOGICALFILENAME,
                       @BUF);
```

Using the INITIALIZER Procedure

The INITIALIZER procedure sets up the SIO FCBs using information from the RUCB and messages read from the $RECEIVE file as shown in Figure 15-2.

Figure 15-2. The INITIALIZER Process


RUCB
Common FCB
Number of File FCBs
Common FCB
Inititalizer
Startup Message
Input File Name
Output File Name
Default Values
Assign Message
Logical File Name
Access Mode
Assign Message
Logical File Name
Block Length
FCB
FCB
FCB
VST070.VSD

The actions of the INITIALIZER procedure are summarized as follows:

1. Reads the RUCB to establish the location of the common FCB (immediately after the RUCB) and the number of FCBs that it will access. (For native callers, the number of FCBs and an array of pointers to the FCBs are passed to INITIALIZER as input parameters.) INITIALIZER verifies the number of FCBs. If the number of FCBs specified when allocating the RUCB and the common FCB, or the number of FCBs specified in the `num^fcbs` parameter, does not match the number of FCBs actually allocated in the program, the process abends.

2. Opens the $RECEIVE file and reads an Open message from the mom process. If INITIALIZER receives an Open message from any process other than its mom process, it replies with error 100. If it receives any other message from a process other than its mom process, it replies with error 60.

3. Reads the Startup message from the $RECEIVE file:

   The INITIALIZER process extracts the input and output file and term names from the Startup message and substitutes them for physical file names in the FCBs whose physical file names were initialized with strings containing #IN, #OUT and #TERM, respectively. Partially qualified names are expanded using the default values also provided in the Startup message.

4. Reads the Assign messages (optional):

   For each Assign message, the INITIALIZER procedure searches each file FCB for the logical file name provided in the Assign message. It then updates all matching FCBs with the information provided in the message.

5. Closes the $RECEIVE file.

Reading Startup Sequence Messages

If your program calls the INITIALIZER procedure to read Assign messages, then it will also read any Param message. However, no special processing of the Param message is done for SIO files. If you want the INITIALIZER to process the Param message, you must provide a procedure to do so, as described in Section 8, Communicating With a TACL Process.

To call the INITIALIZER procedure and read any Assign or Param messages, you provide the name of the RUCB.

For native callers, the basic INITIALIZER procedure call has the form:

```
CALL INITIALIZER(CONTROL^BLOCK,,,,,,,NUM^FCBS,FCB^ARRAY);
```

where NUM^FCBS is the number of FCBs and FCB^ARRAY is an array containing pointers to the FCBs.

The following example shows how to call the INITIALIZER procedure from a native program. The example initializes two FCBs. The example shows the statements needed to set up the input for the INITIALIZER procedure:

```
LITERAL NUM^FCBS = 2;          !Number of FCBs
ALLOCATE^CBS(CONTROL^BLOCK,COMMON^FCB,NUM^FCBS);
ALLOCATE^FCB(IN^FCB,"        #IN             ");
ALLOCATE^FCB(OUT^FCB,"        #OUT            ");
WADDR FCB^ARRAY[0:NUM^FCBS-1]; !Array to hold FCB pointers
.
.
FCB^ARRAY[0] := @IN^FCB;        !Pointer to input FCB
FCB^ARRAY[1] := @OUT^FCB;       !Pointer to output FCB
CALL INITIALIZER(CONTROL^BLOCK,,,,,,,NUM^FCBS,FCB^ARRAY);
```

This form of the INITIALIZER call also works in TAL programs. However, because TAL does not support the WADDR data type, you must include a declarative such as the following in your TAL program:

```
DEFINE WADDR INT #;
```

For TNS callers, the basic INITIALIZER call has the form:

```
CALL INITIALIZER(CONTROL^BLOCK);
```

To call the INITIALIZER procedure without reading Assign or Param messages, use the following call.

Native callers:

```
FLAGS := 0;
FLAGS.<11> := 1;
CALL INITIALIZER(CONTROL^BLOCK,
                 !passthru!,
                 !startupproc!,
                 !paramsproc!,
                 !assignproc!,
                 FLAGS,
                 !timelimit!,
                 NUM^FCBS,
                 FCB^ARRAY);
```

TNS callers:

```
FLAGS := 0;
FLAGS.<11> := 1;
CALL INITIALIZER(CONTROL^BLOCK,
                 !passthru!,
                 !startupproc!,
                 !paramsproc!,
                 !assignproc!,
                 FLAGS);
```

Setting bit 11 of the *`flags`* parameter to 1 inhibits reading of the Assign and Param messages.

Note. When using INITIALIZER with SIO procedures, the setting up of FCBs with the information contained in the Startup and Assign messages is automatic. You do not need to provide user-written procedures to process the Startup and Assign messages as you would in a non-SIO environment (see Section 8, Communicating With a TACL Process) unless you want to perform additional processing of these messages.

Setting Up File Access

The following paragraphs describe how to set up the characteristics that control access to an SIO file. You will learn how to set the access mode, exclusion mode, record length, file code, extent sizes, and block length.

You set file characteristics by putting information into the FCB. There are two ways to do this:

- The user of your program can use ASSIGN commands to set file characteristics at run time. Your program must call INITIALIZER to accept Assign messages and give the file a logical file name using the ASSIGN^LOGICALFILENAME option of the SET^FILE call.
- You can set the file characteristics programmatically using calls to SET^FILE. File characteristics set this way override assignments made by reading Assign messages if the SET^FILE call comes after the INITIALIZER call. Conversely, Assign messages read by INITIALIZER override SET^FILE calls made before the call to INITIALIZER.

Note. Setting these file characteristics is optional. The system provides default values.

If you perform any SET^FILE operation before opening the file and that SET^FILE operation generates an error, then the process abends. The reason for the abend is that you cannot turn off ABORT^XFERERR until you open the file.

Some SET^FILE operations are only accepted before the file is opened; these operations will generate an error if performed after the file is opened. See the *Guardian Procedure Calls Reference Manual* for a list of SET^FILE operations that can be done only before opening the file.

Specifying the File Access Mode

You can specify the access mode for an SIO file as read/write, read only, or write only.

Use the ASSIGN command to set the file access mode at run time. You can specify the access mode as read/write, read only, or write only by setting the *`access-spec`* parameter to I-O, INPUT, or OUTPUT, respectively. The following example assigns read-only access to the file with the logical name INPUT^FILE:

```
1>  ASSIGN INPUT^FILE,,INPUT
```

Use the SET^FILE ASSIGN^OPENACCESS operation to programmatically set the access mode. You set the access mode by setting the *`new-value`* parameter to READWRITE^ACCESS, READ^ACCESS, or WRITE^ACCESS.

The following example sets the access mode to read only for the file associated with the INFILE FCB:

```
CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,READ^ACCESS);
```

If the access mode is not specified, its default value depends on the file type as follows:

File Type	**Access Mode**
Operator process	Read/write
Process	Read/write
$RECEIVE	Read/write
Disk file	Read/write
Terminal	Read/write
Printer	Write
Magnetic tape	Read/write

For more details about access modes, refer to Section 3, Coordinating Concurrent File Access.

Specifying the Exclusion Mode

You can set the exclusion mode of a file to shared, exclusive, or protected.

Use the ASSIGN command to set the exclusion mode at run time. You specify the exclusion mode by setting the *exclusion-spec* parameter to SHARED, EXCLUSIVE, or PROTECTED. The following example assigns shared access to the file associated with the logical name INPUT^FILE:

```
2>  ASSIGN INPUT^FILE,,SHARED
```

Use the SET^FILE ASSIGN^OPENEXCLUSION operation to programmatically set the exclusion mode. You set the exclusion mode by setting the *new-value* parameter to SHARE, EXCLUSIVE, or PROTECTED. The following example sets the exclusion mode to shared for the file associated with the INFILE FCB:

```
CALL SET^FILE(INFILE,ASSIGN^OPENEXCLUSION,SHARE);
```

If the exclusion mode is not specified, its default value depends on the access mode as follows:

Access Mode	**Exclusion Mode**
Read only	Shared mode for terminals, otherwise protected mode
Write only	Shared mode for terminals, otherwise exclusive mode
Read/write	Shared mode for terminals, otherwise exclusive mode

For more details about exclusion modes, refer to Section 3, Coordinating Concurrent File Access.

Specifying the Logical-Record Length

Use the ASSIGN REC command to specify the logical-record length at run time. The *`record-size`* parameter specifies the length. The following example specifies a logical-record length of 256 bytes:

```
3>  ASSIGN INPUT^FILE,,REC 256
```

Use the SET^FILE ASSIGN^RECORDLENGTH operation to programmatically set the logical-record length. You set the *`new-value`* parameter to the number of bytes. The following example sets a logical-record length of 256 bytes:

```
RECORD^LENGTH := 256;
CALL SET^FILE(INFILE,
              ASSIGN^RECORDLENGTH,
              RECORD^LENGTH);
```

If you do not specify the record length, the system provides a default value of 132 bytes for all file types except disk files. The default record length for a disk file is set when you create the file.

Section 5, Communicating With Disk Files, discusses file records in detail.

Specifying the File Code

You can assign an application-dependent file code to a file. Setting the file code affects the subsequent OPEN^FILE call as follows:

- If the file already exists, the specified file code must match the code of the existing file or the open will fail.
- If the file does not already exist and the OPEN^FILE procedure is called with the AUTO^CREATE flag set, then a file is created with the specified file code.

Codes in the range 100 through 999 are reserved.

Set the file code at run time using the ASSIGN CODE command. The *`file-code`* parameter specifies the file code. The following example sets the file code to 101, the code for an EDIT file:

```
4>  ASSIGN LFILE,, CODE 101
```

Set the file code programmatically using the SET^FILE ASSIGN^FILECODE operation. The following example also assigns code 101 to a file:

```
LITERAL EDIT^FILE = 101;
 .
 .
CALL SET^FILE(DFILE,
              ASSIGN^FILECODE,
              EDIT^FILE);
```

If you do not specify a file code, the system assigns a file code to the file. The SIO-assigned file code is 101 if you have supplied a block buffer of at least 1024 bytes; it is 0 otherwise.

Specifying Extent Sizes

For disk SIO files, you can set the primary and secondary extent sizes only if the subsequent OPEN^FILE call will create the file (see Opening and Creating SIO Files, later in this section). If the file already exists, then the new extent size is ignored. For a general discussion of extents, refer to Section 2, Using the File System.

The size of primary and secondary extents can vary from 1 through 4000 pages in increments of one page, where a page is 2048 bytes.

You can set the primary or secondary extent sizes at run time using the ASSIGN command with the EXT option. The following example sets the primary extent to 8 megabytes and each of the secondary extents to one megabyte:

```
5>  ASSIGN LFILE,,EXT(4000,500)
```

Set the extent sizes programmatically using the SET^FILE ASSIGN^PRIMARYEXTENTSIZE and ASSIGN^SECONDARYEXTENTSIZE operations. The following examples set primary and secondary extent sizes:

```
PRIMARY^EXTENT^SIZE := 4000;
SECONDARY^EXTENT^SIZE := 500;
CALL SET^FILE(DFILE,
              ASSIGN^PRIMARYEXTENTSIZE,
              PRIMARY^EXTENT^SIZE);

CALL SET^FILE(DFILE,
              ASSIGN^SECONDARYEXTENTSIZE,
              SECONDARY^EXTENT^SIZE);
```

If you do not specify extent sizes, the default values are 8 pages for the primary extent and 32 pages for each secondary extent. The maximum number of extents is 500.

Specifying the Physical-Block Length

The physical-block length is the number of bytes transferred between the file and the process in one I/O operation. You indicate blocking by setting the physical-block length. You must set the physical-block length when accessing a file in EDIT format.

A physical block is made up of one or more records. If the block length is not exactly divisible by the record length, then the portion of the block following the last record is filled with blanks.

You can set the physical-block length at run time using the ASSIGN command with the BLOCK option. The following example sets the block length to 2048 bytes:

```
6>  ASSIGN LFILE,,BLOCK 2048
```

You can also set the physical-block length programmatically using the SET^FILE ASSIGN^BLOCKLENGTH operation. The following example also sets the block length to 2048 bytes:

```
BLOCK^LENGTH := 2048;
CALL SET^FILE(DFILE,
              ASSIGN^BLOCKLENGTH,
              BLOCK^LENGTH);
```

If you do not specify a block length, then no blocking is performed.

Reassigning a Physical File Name to a Logical File

You can use an ASSIGN command to reassign a physical file name at run time. To do this, you must have already associated a logical file name with the FCB as described earlier in this section under Assigning a Logical File Name.

The following example reassigns the physical file name and sets the physical-block size:

```
7>  ASSIGN LFILE,DATA1,BLOCK 2048
```

If you do not reassign a physical file name, then the FCB retains its association with the file name set up by the ALLOCATE^FCB or ALLOCATE^FCB^D00 DEFINE.

Sample Initialization

The following procedure performs initialization for some SIO files: the input file, the output file, and an additional disk file.

The names of the input and output files are delivered to the process through the Startup message. The procedure checks to see whether these names refer to the same process or terminal file. If so, then the file is assigned read/write access. If the input and output files are different, then the input file is assigned read-only access and the output file write-only access.

The access mode of the disk file is not assigned. The user of the program can assign the access mode using an ASSIGN command.

Note. This example is written to execute in both the native and the TNS environments. The CALL_SET^FILE_ADDRESS_ and CALL_CHECK^FILE_ADDRESS_ DEFINEs are used to select the appropriate SET^FILE and CHECK^FILE calls, respectively. The same form of the INITIALIZER call is used for both environments.

```
?INSPECT, SYMBOLS
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST

!Allocate the RUCB and the common FCB:
ALLOCATE^CBS(RUCB,COMMON^FCB,3);

!Allocate an FCB for each SIO file:
ALLOCATE^FCB(INFILE,"        #IN             ");
ALLOCATE^FCB(OUTFILE,"        #OUT            ");
ALLOCATE^FCB(DFILE,"$OURVOL MYSUBVOLDATAFILE");

!The following DEFINE is required because TAL does not
!support type WADDR
?IFNOT PTAL
DEFINE WADDR INT # ;
?ENDIF PTAL

!Set up input for INITIALIZER call:
LITERAL NUM^FCBS = 2;              !number of FCBs
WADDR FCB^ARRAY[O:NUM^FCBS-1];     !pointers to FCBs

LITERAL PROCESS = 0,               !identify process file
        TERMINAL = 6;              !identify terminal file

INT DEVICE^TYPE,                   !type of device
    PHYS^REC^LEN,                  !length of physical record
    INTERACTIVE;                   !set if input file same as
                                   !output file

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(SET^FILE,INITIALIZER,
?                       CHECK^FILE,DEVICEINFO,FNAMECOMPARE,
?                       PROCESS_STOP_)
?LIST
```

```
!-------------------------------------------------------------
!Procedure for initializing all SIO files used by this
!application.
!-------------------------------------------------------------

PROC INITIALIZE^FILES;
BEGIN

   INT  .INFNAME;                  !Input file name
   INT  .OUTFNAME;                 !Output file name
   INT   ERR;
   INT  .BUF[0:39];                !Contains logical file
                                   !name
   STRING .SBUF := @BUF '<<' 1;    !String pointer to BUF

!  Assign a logical file name to each SIO file:

   SBUF ':=' [5, "INPUT"];
   CALL_SET^FILE_ADDRESS_(ERR,INFILE,ASSIGN^LOGICALFILENAME,
                          @BUF);

   SBUF ':=' [6,"OUTPUT"];
   CALL_SET^FILE_ADDRESS_(ERR,OUTFILE,ASSIGN^LOGICALFILENAME,
                          @BUF);

   SBUF ':=' [4,"DATA"];
   CALL_SET^FILE_ADDRESS_(ERR,DFILE,ASSIGN^LOGICALFILENAME,
                          @BUF);

!  Initialize the FCBs:

   FCB^ARRAY[0] := @IN^FCB;
   FCB^ARRAY[1] := @OUT^FCB;
   CALL INITIALIZER(RUCB,,,,,,,NUM^FCBS,FCB^ARRAY);

!  Get the physical file names for the input and output
!  files:

   CALL_CHECK^FILE_ADDRESS(ERR,INFILE,FILE^FILENAME^ADDR,
                           @INFNAME);
   CALL_CHECK^FILE_ADDRESS(ERR,OUTFILE,FILE^FILENAME^ADDR,
                           @OUTFNAME);

!  Find out whether the input and output files are the same
!  and therefore used interactively.  This may apply to a
!  terminal or a process:

   CALL DEVICEINFO(INFNAME,DEVICE^TYPE,PHYS^REC^LEN);
   INTERACTIVE :=
      IF (DEVICE^TYPE.<4:9> = TERMINAL OR
      DEVICE^TYPE.<4:9> = PROCESS)
      AND NOT FNAMECOMPARE(INFNAME,OUTFNAME)
      THEN -1 ELSE 0;
```

```
!  If interactive, then set up the access mode for the input
!  file for reading and writing.  There is no need to set
!  the output file because it refers to the same file:

   IF INTERACTIVE THEN
      CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,
                    READWRITE^ACCESS)

!  If the the input and output files are different, then set
!  up the access mode for the input file as read only and
!  the access mode of the output file as write only:

   ELSE BEGIN
      CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,
                    READ^ACCESS);
      CALL SET^FILE(OUTFILE,ASSIGN^OPENACCESS,
                    WRITE^ACCESS);
   END;
END;
```

Opening and Creating SIO Files

This subsection shows how to use the OPEN^FILE procedure to open SIO files. In addition to making files available to the SIO procedures that perform I/O, the OPEN^FILE procedure also sets up many file characteristics. This subsection shows how to perform the following functions:

- Open an SIO file
- Create the SIO file at the same time as opening it
- Enable block buffering and size the block buffer
- Purge the data of an SIO file on opening—you need write access to do this

In addition to the functions listed above, you can set many other file-transfer characteristics while opening an SIO file. Later parts of this section show how to perform many of these other operations, for example:

- Specify blank-padded records: see Reading and Writing SIO Files.
- Open a file for nowait I/O: see Handling Interprocess Messages.
- Specify the error file: see Handling SIO Errors.

Note. You must apply file characteristics to each file using the SET^FILE procedure before each file open. If you close an SIO file and then reopen it without reapplying the file characteristics, then the SIO routines set the default values for the characteristics.

Setting Flag Values in the OPEN^FILE Call

You set many of the file-transfer characteristics at file-open time using flag values passed to the OPEN^FILE procedure. Two optional parameters, *`flags`* and *`flags-mask`*, allow you to set or clear flag values. Each bit in the *`flags`* parameter represents a flag value; each corresponding bit in *`flags-mask`* specifies whether the supplied flag value should be assigned to the file. The GPLDEFS file provides literals to make flag-setting easy.

To set a flag, set the corresponding bit in both *`flags`* and *`flags-mask`*. For example:

```
FLAGS := AUTO^TOF;
FLAGS^MASK := AUTO^TOF;
CALL OPEN^FILE(COMMON^FCB,
               OUTFILE,
               !block^buffer!,
               !block^bufferlen!,
               FLAGS,
               FLAGS^MASK);
```

To clear a flag, set the flag bit in *`flags-mask`* but not in *`flags`* (because the *`flags`* parameter defaults to 0). For example:

```
FLAGS^MASK := AUTO^TOF;
CALL OPEN^FILE(COMMON^FCB,
               OUTFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags!,
               FLAGS^MASK);
```

If AUTO^TOF is not specified in *`flags-mask`*, then the flag value for the file is unchanged regardless of the value in *`flags`*. Refer to the *Guardian Procedure Calls Reference Manual* for a complete list of flag values.

Opening SIO Files: Simplest Form

To open an SIO file, you supply the OPEN^FILE procedure with the names of the file FCB and the common FCB. The operating system responds as follows:

- Opens the file specified by FCB name. The file attributes depend on how the attributes were set in the FCB by the SET^FILE calls or Assign messages.
- Associates the file with the common FCB.

The following example shows the use of the OPEN^FILE procedure in its simplest form to open the input file:

```
CALL OPEN^FILE(COMMON^FCB,
               INFILE);
```

Creating SIO Files

You can use the OPEN^FILE procedure to create and open a file. To do so you must perform the following operations:

1. Set up the file FCB with write-only access.
2. Issue an OPEN^FILE procedure call with the AUTO^CREATE flag set. The AUTO^CREATE flag is set by default.

The following example creates an SIO file:

```
CALL SET^FILE(OUTFILE,
              ASSIGN^OPENACCESS,
              WRITE^ACCESS);
.
.
FLAGS := AUTO^CREATE;
FLAGS^MASK := AUTO^CREATE;
CALL OPEN^FILE(COMMON^FCB,
               OUTFILE,
               !block^buffer!,
               !block^bufferlen!,
               FLAGS,FLAGS^MASK);
```

If the file already exists, then the above open will proceed, ignoring the request to create the file. You can, however, cause the open to fail if the file already exists by setting the MUSTBENEW flag. If the MUSTBENEW flag is set when you attempt to create a file, and if the file already exists, the OPEN^FILE call returns error 10:

```
CALL SET^FILE(OUTFILE,
              ASSIGN^OPENACCESS,
              WRITE^ACCESS);
.
.
FLAGS := AUTO^CREATE + MUSTBENEW;
FLAGS^MASK := AUTO^CREATE + MUSTBENEW;
ERROR := OPEN^FILE(COMMON^FCB,
                   OUTFILE,
                   !block^buffer!,
                   !block^bufferlen!,
                   FLAGS,
                   FLAGS^MASK);
```

Note that the MUSTBENEW flag is effective only if the AUTO^CREATE flag is also set and the file FCB is set up for write-only access. If either of these conditions is not true, then the open proceeds.

By default, the AUTO^CREATE flag is set but the MUSTBENEW flag is not set.

To create a file with an access mode other than write only, you must create the file with write-only access as described above, close the file, change the access permissions with SET^FILE, and then reopen the file.

Block Buffering With SIO Files

You can specify a block buffer when you open an SIO file.

△ **Caution.** Do not mix regular Guardian procedure calls with SIO procedure calls on a file for which you are providing a block buffer. You might corrupt your data.

The use of the block buffer depends on the type of file you are opening:

- For structured Enscribe files, you use sequential block buffering to improve the efficiency of read operations. You simply set the buffer size. See the *Enscribe Programmer’s Guide* for details.
- For EDIT files, the block buffer must provide space for the EDIT file pages as they are assembled or disassembled. See Accessing EDIT Files later in this section.
- For SIO files that are neither Enscribe files nor EDIT files, the buffer is used for record blocking and unblocking. For record blocking and unblocking to happen, your program must perform the following operations:
  - Supply the block buffer name and length to the OPEN^FILE procedure call
  - Size the block buffer to contain at least one logical record
  - Set the access mode in the FCB to read only or write only

2 If a block buffer is used in a TNS caller, it must be located in G[0:32767] of the data area. This limitation does not apply to native callers. See Section 16, Creating and Managing Processes, for details.

The following example specifies a block buffer named OUTBLKBUF that is 4096 bytes long:

```
LITERAL OUTBUFLEN = 4096;
INT .OUTBLKBUF[0:OUTBUFLEN -1];
  .
  .
CALL OPEN^FILE(COMMON^FCB,
               OUTFILE,
               OUTBLKBUF,
               OUTBUFLEN);
```

Purging Data When Opening

You can delete all data from an SIO file when you open it by setting the PURGE^DATA flag in the OPEN^FILE call. The file FCB must be set up for write-only access.

The following example purges the data from the file with FCB DFILE:

```
CALL SET^FILE(DFILE,
              ASSIGN^OPENACCESS,
              WRITE^ACCESS);
  .
  .

FLAGS := PURGE^DATA;
FLAGS^MASK := PURGE^DATA;
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               FLAGS,
               FLAGS^MASK);
```

Getting Information About SIO Files

Use the CHECK^FILE procedure to obtain information about a given SIO file. The CHECK^FILE procedure gets its information from the FCB of the file and can include:

- File attributes set when the file was opened or explicitly set by the SET^FILE procedure
- General status information about the file that is dynamically updated by the SIO procedures that use the file

An example of retrieving attribute information might be to get the name of the file. The FILE^FILENAME^ADDR operation returns the address of the name. Note that for CHECK^FILE calls that return an address value, the TAL and pTAL forms of the call differ.

pTAL example:

```
CALL CHECK^FILE(INFILE,
                FILE^FILENAME^ADDR,
                @INFNAME);
```

TAL example:

```
@INFNAME := CHECK^FILE(INFILE,
            FILE^FILENAME^ADDR);
```

One example of file status that you might want to retrieve is whether an I/O operation is outstanding on a file. You may need this information, for example, before calling WAIT^FILE to check for completion of an I/O operation. The FILE^LOGIOOUT operation returns this information:

```
OUTSTANDING := CHECK^FILE(INFILE, FILE^LOGIOOUT);
```

In the above example, CHECK^FILE returns 1 in the integer variable OUTSTANDING if there is an outstanding write operation and returns 2 if there is an outstanding read operation on the file. If there is no I/O outstanding, CHECK^FILE returns 0. This example is typical of the way CHECK^FILE returns status information.

Refer to the *Guardian Procedure Calls Reference Manual* for details on all operations possible with the CHECK^FILE procedure call as well as syntax differences between the TAL and pTAL versions.

Reading and Writing SIO Files

The READ^FILE procedure allows you to read records from an SIO file; the WRITE^FILE procedure allows you to write records to a file. In addition, you can use options to the SET^FILE and OPEN^FILE procedures to affect the way you perform I/O with SIO procedures. This subsection explains how to use these procedures to perform the following operations:

- Read records from an SIO file using the READ^FILE procedure.
- Write records to an SIO file using the WRITE^FILE procedure.
- Change the interactive read prompt using the OPEN^FILE, SET^FILE, or READ^FILE procedure.
- Handle write operations that are longer than a logical record. Use either the WRITE^FOLD flag of the OPEN^FILE procedure or the SET^WRITE^FOLD operation of the SET^FILE procedure.
- Handle blank padding on reading and writing for records. Use either the WRITE^PAD, WRITE^TRIM, or READ^TRIM flags of the OPEN^FILE procedure, or the SET^WRITE^PAD, SET^WRITE^TRIM, or SET^READ^TRIM operations of the SET^FILE procedure.
- Apply forms control to a printer using the `forms-control-code` parameter of the WRITE^FILE procedure.

Refer to Handling Nowait I/O, later in this section, for information about reading and writing to SIO files opened for nowait I/O.

Refer to Handling Interprocess Messages for information on how to read and write to the $RECEIVE file using SIO procedures.

Handling Basic I/O With SIO Files

The SIO routines permit only sequential access to files (except when reading files in EDIT format). There is no support for positioning within a file to achieve random access. The initial position for sequential access therefore depends on how the file pointers are set up when the file is opened. The access mode determines the first record accessed as follows:

- For read-only access, the initial position for sequential access is the first record in the file. Successive read operations then step through the file one record at a time. For key-sequenced files, record retrieval is in key sequence. For all other file types, record retrieval is in physical-record order.
- For write-only access, the initial position is the end of the file. Each write operation therefore appends a record to the file.
- For read/write access, the initial position is the first record in the file. If the first access is a read operation, then the first record is read and the file pointers move on to the second record for the next read or write operation. If the first access is a write operation, then the first record gets overwritten and the file pointers move on to the next record.

The following paragraphs describe how to use the READ^FILE and WRITE^FILE procedures to perform I/O operations.

Reading Records With SIO Procedures

To read a record from a file, you must supply the READ^FILE procedure with the names of the FCB of the file you wish to read from and of the buffer to receive the record.

The following example reads one record from the input file. In addition to placing the record into the read buffer, this call also returns the number of bytes read in the third parameter:

```
CALL READ^FILE(INFILE,
               READ^BUFFER,
               BYTES^READ);
```

Writing Records With SIO Procedures

To write a record to a file, you must supply the WRITE^FILE procedure with the name of the FCB of the file to write to and of the buffer that contains the record to be written:

```
CALL WRITE^FILE(OUTFILE,
                WRITE^BUFFER,
                WRITE^COUNT);
```

Changing the Interactive Read Prompt

Normally, a READ^FILE call against a terminal file displays a question mark (?) character on the terminal to prompt for input. (Internally, the SIO routine is actually executing a WRITEREAD procedure call.) You can change the prompt to one of the following:

- An alternate character specified in the *`prompt-char`* parameter of the OPEN^FILE procedure
- An alternate character specified by a SET^FILE SET^PROMPT operation
- A string of characters supplied to the READ^FILE procedure

The following example uses the OPEN^FILE procedure to change the interactive prompt to a colon:

```
PROMPT^CHAR ':=' ":";
CALL OPEN^FILE(COMMON^FCB,
               INPUT,
               INBLKBUF,
               INBLKLEN,
               !flags!,
               !flags^mask!,
               !max^recordlen!,
               PROMPT^CHAR);
```

The next example does the same using the SET^FILE procedure:

```
NEW^PROMPT ':=' ":";
CALL SET^FILE(INPUT,
              SET^PROMPT,
              NEW^PROMPT);
```

To issue a multiple-character prompt, you set the prompt in the read buffer and then issue the READ^FILE procedure call. The *`prompt-count`* parameter specifies how many characters to write to the terminal from the read buffer. For example:

```
BUFFER ':=' "Press 'y' or 'n': ";
PROMPT^COUNT := 18;
CALL READ^FILE(INPUT,
               BUFFER,RCOUNT,PROMPT^COUNT);
```

Note that the prompt is overwritten by the text returned in the read buffer.

Handling Long Writes

The OPEN^FILE and SET^FILE procedures provide options that allow write operations that are longer than the logical-record length without losing any data.

The SIO routines normally truncate write operations that are longer than the logical-record length, saving only that part of the write that fits into the logical record. In such cases, the system does not issue an error. However, if you issue the OPEN^FILE procedure with the WRITE^FOLD flag set or the SET^FILE procedure with the SET^WRITE^FOLD operation, then the write operation will be split into as many logical records as it needs.

The following example shows how you can use the OPEN^FILE procedure to write a 42-character buffer to a file whose logical-record length is 16 bytes. The data is saved in the file in three logical records:

```
RECORD^LENGTH := 16;
CALL SET^FILE(DFILE,
              ASSIGN^RECORDLENGTH,
              RECORD^LENGTH);

CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               WRITE^FOLD,
               WRITE^FOLD);
 .
 .
BUFFER ':=' "This write operation is more than 16 bytes"
            -> @S^PTR;
CALL WRITE^FILE(DFILE,
                BUFFER,
                @S^PTR '-' @SBUFFER);
```

The next example uses the SET^FILE procedure to permit long writes. Here, the write-fold feature is turned on just before a record is written to the file. Then it is turned off again after the record is written. Subsequent writes are then limited to one logical record.

```
LITERAL ON = 1;
LITERAL OFF = 0;
RECORD^LENGTH := 16;
CALL SET^FILE(DFILE,
              ASSIGN^RECORDLENGTH,
              RECORD^LENGTH);

CALL OPEN^FILE(COMMON^FCB,
               DFILE);
 .
 .
BUFFER ':=' "This write operation is more than 16 bytes"
            -> @S^PTR;
CALL SET^FILE(DFILE,
              SET^WRITE^FOLD,
              ON);

CALL WRITE^FILE(DFILE,
                BUFFER,
                @S^PTR '-' @SBUFFER);

CALL SET^FILE(DFILE,
              SET^WRITE^FOLD,
              OFF);
```

By default, the write-fold feature is turned off.

When reading records that were written using the write-fold feature, you must issue one READ^FILE procedure call for each logical record.

Handling Padding Characters

Several options to the OPEN^FILE and SET^FILE procedures can affect the way blank padding is applied to records written and read by SIO procedures.

The SIO routines support the following operations:

- Trimming trailing blanks from a logical record before writing
- Padding short logical records with blanks before writing
- Trimming trailing blanks from logical records after reading

The following paragraphs explain how to accomplish these operations using SIO procedures.

Trimming Trailing Blanks Before Writing

Use the WRITE^TRIM flag with the OPEN^FILE procedure or the SET^FILE SET^WRITE^TRIM operation to remove trailing blanks from a record before writing the record to a file. By default, write-trailing-blank-trimming is turned on.

The following statement uses the OPEN^FILE procedure to turn off write-trailing-blank-trimming because WRITE^TRIM is selected in the flag mask while the *flags* parameter defaults to zeros:

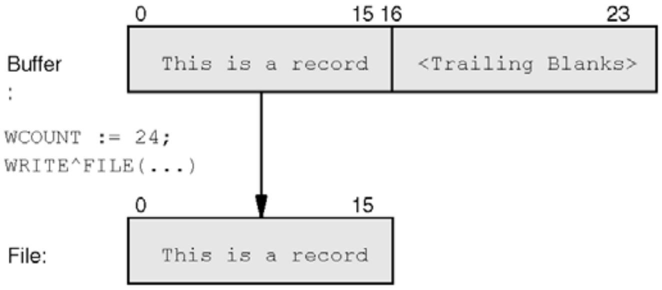
```
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags parameter!,
               WRITE^TRIM);
```

The following statement turns it back on again using SET^FILE:

```
LITERAL ON = 1;
CALL SET^FILE(DFILE,
              SET^WRITE^TRIM,
              ON);
```

In the following example, the WRITE^FILE procedure would write 24 bytes of data into the logical record in the data file if write-trailing-blank-trimming was turned off. However, because write-trailing-blank-trimming is turned on, trailing blanks are not copied to the file. Therefore only 16 bytes are actually written:

```
LITERAL ON = 1;
RECORD^LENGTH := 24;
CALL SET^FILE(DFILE,
              ASSIGN^RECORDLENGTH,
              RECORD^LENGTH);

CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags parameter!,
               WRITE^TRIM);
 .
 .
CALL SET^FILE(DFILE,
              SET^WRITE^TRIM,
              ON);

WRITE^BUFFER[0] ':=' "  ";
WRITE^BUFFER[1] := WRITE^BUFFER[0] FOR 23;
WRITE^BUFFER ':=' "This is a record";
CALL WRITE^FILE(DFILE,
                WRITE^BUFFER,
                RECORD^LENGTH);
```



VST071.VSD

Padding Short Records With Blanks When Writing

Use the WRITE^PAD flag with the OPEN^FILE procedure or the SET^FILE SET^WRITE^PAD operation to pad a short record with blanks before writing the record to a file. By default, write-blank-padding is turned on for disk files with fixed-length records and turned off for all other files.

The following statement uses the OPEN^FILE procedure to turn on write-blank-padding:

```
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               WRITE^PAD,
               WRITE^PAD);
```

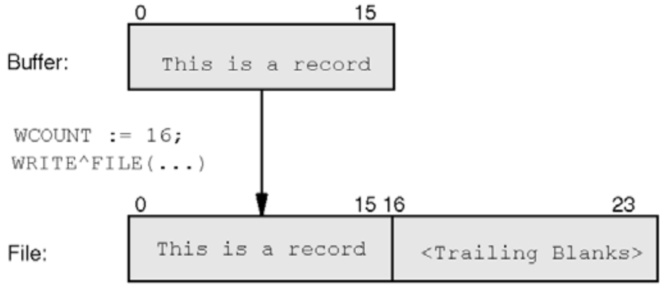
The following statements have the same effect using SET^FILE:

```
LITERAL ON = 1;
 .
 .
CALL SET^FILE(DFILE,
              SET^WRITE^PAD,
              ON);
```

In the next example, the WRITE^FILE procedure would normally write 16 bytes of data into the logical record in the data file of variable-length records. However, because the WRITE^PAD flag is set in the OPEN^FILE call, the record gets padded with trailing blanks up to the length of the logical record:

```
RECORD^LENGTH := 24;
CALL SET^FILE(DFILE,
              ASSIGN^RECORDLENGTH,
              RECORD^LENGTH);
```

```
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               WRITE^PAD,
               WRITE^PAD);
 .
 .
WRITE^BUFFER ':=' "This is a record" -> @S^PTR;
CALL WRITE^FILE(DFILE,
                WRITE^BUFFER,
                @S^PTR '-' @SWRITE^BUFFER);
```



VST072.VSD

Trimming Trailing Blanks on Reading

By default, trailing blanks are trimmed from records after reading. You can turn this feature on or off using the READ^TRIM flag with the OPEN^FILE procedure or the SET^FILE SET^READ^TRIM operation. When read-trailing-blank-trimming is turned on and the READ^FILE procedure reads a record with trailing blanks, the entire record is read into the read buffer, including the blanks. However, the count of bytes read returned by the READ^FILE procedure counts only those characters that precede the first of the trailing blanks.

If read-trailing-blank-trimming is turned off, the count of bytes read indicates the length of the entire record.

The following statement uses the OPEN^FILE procedure to turn off read-trailing-blank-trimming:

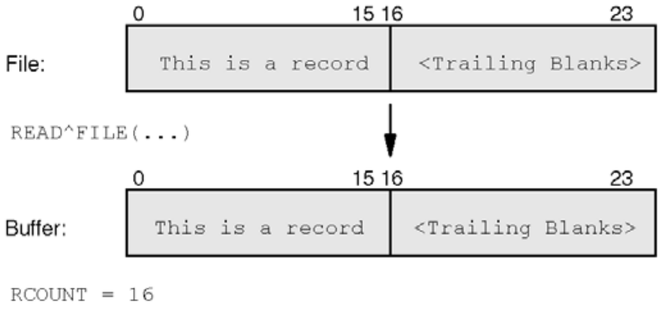
```
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags parameter!,
               READ^TRIM);
```

The following statements turn it back on again using SET^FILE:

```
LITERAL ON = 1;
 .
 .
CALL SET^FILE(DFILE,
              SET^READ^TRIM,
              ON);
```

In the next example, the READ^FILE call with read-trailing-blank-trimming turned off would indicate that 24 bytes have been read (the length of the record). However, because read trailing-blank-trimming is turned on, the READ^FILE call returns a read count of only 16 bytes:

```
LITERAL ON = 1;
RECORD^LENGTH := 24;
CALL SET^FILE(DFILE,
               ASSIGN^RECORDLENGTH,
               RECORD^LENGTH);

CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags parameter!,
               READ^TRIM);
 .
 .
CALL SET^FILE(DFILE,
              SET^READ^TRIM,
              ON);

CALL READ^FILE(DFILE,
               READ^BUFFER,
               BYTES^READ);
```



VST073.VSD

Writing to a Printer

A program that uses SIO procedures can control or write to a printer in a way similar to programs that use regular Guardian procedure calls. Such programs can use the printer control language (PCL) as described in Section 11, Communicating With Printers, and in the user guide for your printer.

A program can also enable level-3 spooling through SIO. Level-3 spooling uses the spooler interface procedures to improve performance when writing to a spooler collector by using buffered I/O. (Refer to the description of the OPEN^FILE procedure in the *Guardian Procedure Calls Reference Manual*.)

Communicating with a printer using SIO procedure calls has some differences from communicating using regular Guardian procedure calls. These differences affect the following operations:

- Moving the printer automatically to the top of the form when you open it
- Accessing control functions through the WRITE^FILE procedure instead of the CONTROL procedure
- Issuing PCL commands using the WRITE^FILE procedure

The following paragraphs explain how to perform these operations using SIO procedure calls.

Opening a Printer and Issuing an Automatic Top of Form

To access a printer using SIO procedures, you must open the printer file using the OPEN^FILE procedure. By default, the SIO routines position the printer at the top of the form if it is opened for write access (but not read/write access):

```
CALL OPEN^FILE(COMMON^FCB,
               PFILE);
```

Use the AUTO^TOF flag in the flags mask without setting the corresponding bit in the *flags* parameter if you do not want to be automatically moved to the top of the form:

```
CALL OPEN^FILE(COMMON^FCB,
               PFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags parameter!,
               AUTO^TOF);
```

Issuing CONTROL Functions

Use the *`forms-control-code`* parameter of the WRITE^FILE procedure to provide forms control for the printer. This parameter can take any value that is valid for the second parameter of CONTROL operation 1. You should not, however, try to use the CONTROL procedure; instead use WRITE^FILE.

The following example issues a form feed before writing the contents of the write buffer to the file:

```
LITERAL FORM^FEED = 0;
  .
  .
CALL WRITE^FILE(PFILE,
                BUFFER,
                WCOUNT,
                !reply^error^code!,
                FORM^FEED);
```

Refer to the *Guardian Procedure Calls Reference Manual* for a complete list of values for the second parameter to CONTROL operation 1.

Issuing PCL Commands

You can issue any given PCL command supported by a given printer using the WRITE^FILE procedure call. You simply supply the appropriate escape sequence in the write buffer. The following example sets the left and right margins:

```
WRITE^BUFFER ':=' [%33,"&a99m9L"] -> @S^PTR;
CALL WRITE^FILE(PFILE,
                WRITE^BUFFER,
                @S^PTR '-' @SWRITE^BUFFER);
```

Refer to Section 11, Communicating With Printers, for a description of PCL and some commonly used escape sequences.

Accessing EDIT Files

SIO procedures provide one programmatic way to write to files in EDIT format. The other way is to use the IOEdit procedures described in Section 14, Using the IOEdit Procedures.

Using SIO procedures, access to an EDIT file is like access to any other SIO file, with the following exceptions:

- You must open an EDIT file in either read-only or write-only mode; any attempt to open an EDIT file in read/write mode will fail. In addition, the open must specify a block buffer.
- When reading records in an EDIT file, you can save the current position and return to it later.

The following paragraphs explain how to perform these operations.

Opening an EDIT File

The way you open an EDIT file depends on whether you will read from the file or write to the file. Specifically, the minimum size of the block buffer is different for reading and for writing. The block buffer provides space for the EDIT file pages as they are assembled or disassembled.

Writing to an EDIT File

To write to a file in EDIT format, you must open the file with a block buffer of at least 1024 bytes:

```
CALL SET^FILE(EDITFILE,
              ASSIGN^OPENACCESS,
              WRITE^ACCESS);

EDBLKLEN := 1024;
CALL OPEN^FILE(COMMON^FCB,
               EDITFILE,EDBLKBUF,EDBLKLEN);
```

Attempting to open an EDIT file for writing with a block buffer less than 1024 bytes long causes SIO error 518, SIOERR^BUFTOOSMALL.

Reading From an EDIT File

To read from a file in EDIT format, you must open the file with a block buffer of at least 144 bytes:

```
CALL SET^FILE(EDFILE,
              ASSIGN^OPENACCESS,
              READ^ACCESS);

EDBLKLEN := 256;
CALL OPEN^FILE(COMMON^FCB,
               EDFILE,EDBLKBUF,EDBLKLEN);
```

Attempting to open an EDIT file for reading with a block buffer less than 144 bytes long also causes an SIOERR^BUFTOOSMALL error (error 518).

Setting the Read Position

Use the SET^EDITREAD^REPOSITION operation of the SET^FILE procedure to read from a specific position in an EDIT file.

Normally, you read from an EDIT file sequentially, just as from any other file. However, a feature specific to EDIT files allows you to save the current position and then return to it later.

The reposition mechanism works like this. While working on the file, you reach a position that you may want to return to later.

1. Save the current position. You do this by saving the second through fourth words of the block buffer you specified to the OPEN^FILE procedure. These words

contain the current file position in an internal format. Note that the current position points to the start of the record that sequentially follows the last record read.

2. Continue processing the EDIT file.
3. Later when you want to return to the saved position, you simply restore the three words to the block buffer and then reposition the file pointer by issuing the SET^FILE SET^EDITREAD^REPOSITION operation.

The following example demonstrates the above sequence:

```
!Save the current position to return to later:

RETURN^POSITION ':=' EDBLKBUF[1] FOR 3 WORDS;

CALL READ^FILE(EDFILE,
               READ^BUFFER,
               BYTES^READ);
 .
 .
!Additional I/O to the EDIT file moves the current
!file position:

CALL READ^FILE(EDFILE,
               READ^BUFFER,
               BYTES^READ);
 .
 .
!Reposition the file pointer to return to saved position:

EDBLKBUF[1] ':=' RETURN^POSITION FOR 3 WORDS;
CALL SET^FILE(EDFILE,
              SET^EDITREAD^POSITION);

!Read again the record that was read just after saving the
!position:

CALL READ^FILE(EDFILE,
               READ^BUFFER,
               BYTES^READ);
```

Handling Nowait I/O

SIO procedures can access files using waited or nowait I/O. With waited I/O, a process waits for each I/O operation to finish before continuing. With nowait I/O, the process initiates an I/O operation and then continues processing; your process then executes in parallel with the I/O operation. Section 4, Using Nowait Input/Output, provides a detailed discussion of waited and nowait I/O.

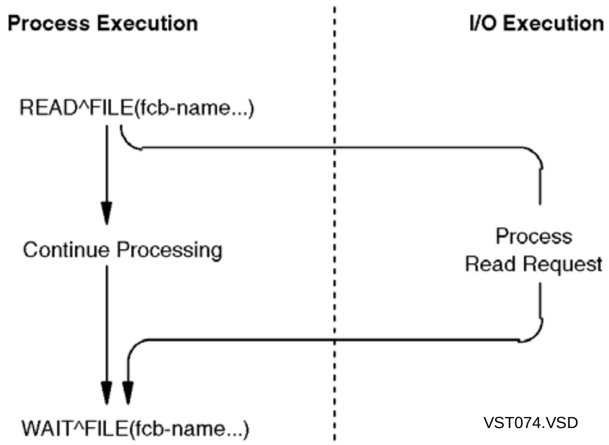
Waiting for One File

In its simplest form, a nowait I/O operation with SIO procedures consists of the following steps:

1. Issue a single SIO operation against one file
2. Continue processing while the I/O takes place
3. Finish the I/O with a call to the WAIT^FILE procedure

Figure 15-3 shows nowait I/O applied to the READ^FILE procedure. The same model also applies to the WRITE^FILE procedure.

Figure 15-3. Nowait I/O Applied to a Single SIO File



To perform nowait I/O on a file, you must open that file specifically for nowait I/O. You do so by setting the NOWAIT flag in the OPEN^FILE procedure call:

```
CALL OPEN^FILE(COMMON^FCB,
               DFILE,
               !block^buffer!,
               !block^bufferlen!,
               NOWAIT,
               NOWAIT);
```

The NOWAIT flag opens the file with a wait depth of 1. This means that you cannot have more than one outstanding I/O operation against one file. Unlike regular Guardian procedures, SIO procedures do not allow you to open files with a wait depth greater than 1.

Note. Unlike file system I/O, SIO allows you to perform waited I/O against a file that was opened for nowait I/O. In the READ^FILE or WRITE^FILE procedure you specify a zero in the *nowait* parameter or omit the parameter. However, you cannot request a nowait operation against a file that is open for waited operations.

Once the file is open, you can issue an I/O operation against it, such as the READ^FILE operation shown below. Note that the sixth parameter must contain a positive number for a nowait operation:

```
CALL READ^FILE(DFILE,
               READ^BUFFER,
               BYTES^READ,
               !prompt^count!,
               !max^read^count!,
               1);
```

Nothing is returned at this point in either the read buffer or the count of bytes read, because the I/O operation has not yet finished.

You complete the I/O operation by calling the WAIT^FILE procedure. You must supply the FCB name, a variable to contain the read count, and a value for the `time-out` parameter. The `time-out` parameter causes WAIT^FILE to respond in one of the following ways:

- Wait indefinitely for the I/O to finish by omitting the `time-out` parameter or setting it to -1D:

  ```
  CALL WAIT^FILE(DFILE,
                 BYTES^READ);
  ```

  On return, BYTES^READ contains the number of bytes transferred by the I/O operation. The buffer specified in the I/O call contains the transferred bytes.

- Return immediately whether the I/O finished or not by setting the `time-out` parameter to zero. This option makes sense only if the WAIT^FILE call is in a loop that gets executed repeatedly until the I/O finishes:

  ```
  LITERAL RETURN^IMMEDIATELY = 0;
   .
   .
  CALL WAIT^FILE(DFILE,
                 BYTES^READ,
                 RETURN^IMMEDIATELY);
  ```

  You can use the FILE^LOGIOOUT operation of the CHECK^FILE procedure to check for completion. If the return value is nonzero, then the I/O is still outstanding. If the return value is zero, then the I/O has finished.

  ```
  OUTSTANDING := CHECK^FILE(DFILE,
                            FILE^LOGIOOUT);
  ```

- Wait until either the I/O finishes or the time-out value expires. You specify the time in one-hundredths of a second. The following example times out after 30 seconds:

  ```
  TIME^LIMIT := 3000D;
  CALL WAIT^FILE(DFILE,BYTES^READ,TIME^LIMIT);
  ```

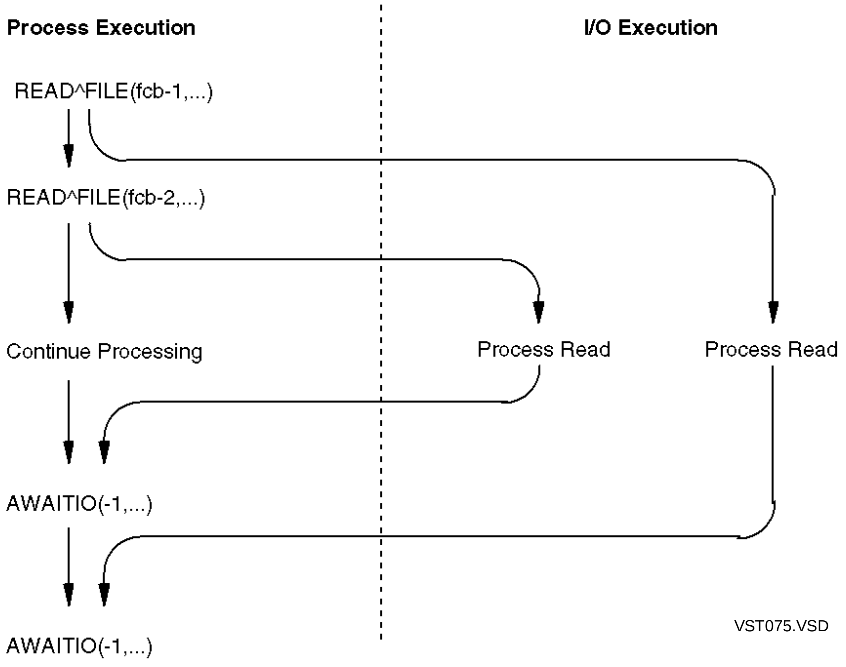
  Again you can use the FILE^LOGIOOUT operation of the CHECK^FILE procedure to find out whether the I/O finished.

Waiting for Any File

You can use SIO procedures to concurrently apply nowait I/O to multiple files. In other words, you can issue nowait I/O operations against more than one file and then wait for any of the I/O operations to finish.

Here, you need to use the AWAITIO procedure as well as the WAIT^FILE procedure to complete the I/O operation. The AWAITIO procedure responds to the first I/O to finish, and then the WAIT^FILE procedure updates the file-state information. Figure 15-4 shows this model (excluding calls to the WAIT^FILE procedure).

Figure 15-4. Nowait I/O Applied to Multiple SIO Files



A typical use for concurrent nowait operations is in a process that communicates with more than one terminal. The process can issue prompts to several users without having to wait for a reply. Instead, the process can continue processing requests from active users. An inactive user can become active again simply by responding to the prompt.

The important steps are outlined as follows:

1. Open each file for nowait I/O using the OPEN^FILE procedure with the NOWAIT flag set. This step is the same as for the single-file model, except that you need to do it once for each file.
2. Establish the file numbers of each of the open files. Nowait I/O applied to multiple files is one of the few occasions when you would mix SIO procedures with regular

Guardian procedures on the same file. You therefore need to obtain the file numbers to be able to access the files using the AWAITIO call.

3. Issue a nowait I/O operation against each file. Here, READ^FILE calls are issued.
4. Issue the AWAITIO call to receive the first I/O operation that finishes. You need to set the file number to -1, indicating that a response from any file will do. On return from the AWAITIO call, the *`file-number`* parameter indicates which I/O finished; the *`count-returned`* parameter contains the number of bytes read, and the data itself is in the buffer indicated in the originating READ^FILE call.
5. Check the file number returned by the AWAITIO call to see which I/O finished, and execute the code appropriate for that file.
6. For the file established in the previous step, you must update the file-state information in the FCB. You have to do this manually, because AWAITIO is not an SIO procedure and therefore does none of this for you. You must update:
   a. The I/O done flag
   b. The number of bytes transferred
   c. The error code

   A combination of SET^FILE and WAIT^FILE procedure calls enables you to do this.

The following TAL example provides skeletal code for waiting for I/O from more than one SIO file:

```
!Open all terminals for nowait I/O:
CALL OPEN^FILE(COMMON^FCB,TERM1,
               !block^buffer!,
               !block^bufferlen!,
               NOWAIT,NOWAIT,
               !max^record^len!,
               !prompt^char!,
               OUTFILE);
CALL OPEN^FILE(COMMON^FCB,TERM2,
               !block^buffer!,
               !block^bufferlen!,
               NOWAIT,NOWAIT,
               !max^record^len!,
               !prompt^char!,
               OUTFILE);

!Establish the file numbers of the open terminal files.  (We
!need these values later for getting information through the
!FILE_GETINFO_ procedure call):
@T1^FNUM := CHECK^FILE(TERM1,FILE^FNUM^ADDR);
@T2^FNUM := CHECK^FILE(TERM2,FILE^FNUM^ADDR);
  .
  .
!Start a read operation on each open terminal file:
CALL READ^FILE(TERM1,READ^BUFFER,BYTES^READ,
               !prompt^count!,
               !max^read^count!,
               1);
CALL READ^FILE(TERM2,READ^BUFFER,BYTES^READ,
               !prompt^count!,
               !max^read^count!,
               1);

!Use the AWAITIO call to receive the first I/O to finish:
FNUM := -1;
CALL AWAITIO(FNUM,
             !buffer^address!,
             RCOUNT);

!If response received from terminal 1:
IF FNUM := T1^FNUM THEN
BEGIN

   !Establish the error value caused by AWAITIO:
   CALL FILE_GETINFO_(T1^FNUM,ERROR);

   !Update the FCB to indicate I/O complete:
   CALL SET^FILE(TERM1,SET^PHYSIOOUT,0);

   !Update the FCB with the number of bytes transferred:
   CALL SET^FILE(TERM1,SET^COUNTXFERRED,RCOUNT);

   !Update the FCB with the error number returned from
   !the FILE_GETINFO_ call following AWAITIO:
   CALL SET^FILE(TERM1,SET^ERROR,ERROR);

   !Update the file-state information:
   CALL WAIT^FILE(TERM1,BYTES^READ);
   !Process the read from terminal 1
     .
     .
END
ELSE
```

```
!Or if response came from terminal 2:
BEGIN

   !Establish the error value returned from AWAITIO:
   CALL FILE_GETINFO_(T2^FNUM,ERROR);

   !Update the FCB to indicate I/O complete:
   CALL SET^FILE(TERM2,SET^PHYSIOOUT,0);

   !Update the FCB with the number of bytes transferred:
   CALL SET^FILE(TERM2,SET^COUNTXFERRED,RCOUNT);

   !Update the FCB with the error number returned from the
   !FILE_GETINFO_ call following AWAITIO:
   CALL SET^FILE(TERM2,SET^ERROR,ERROR);

   !Update the file-state information:
   CALL WAIT^FILE(TERM2,BYTES^READ);
   !Process the read from terminal 2
     .
     .
END;
```

Handling Interprocess Messages

This subsection discusses how processes communicate with each other using SIO procedures. The SIO procedures perform most of the functions provided by the regular Guardian procedures.

As when working with file-system procedures directly, you send a message to another process and wait for a reply. Typically, data is expected in the reply; see Passing Messages and Reply Text Between Processes, later in this subsection. However, you can use a simpler model for sending messages and receiving replies if the receiving process will never send data in the reply; see Passing Messages Between Processes: No Reply Data.

Refer to Section 6, Communicating With Processes, for details of interprocess communication (IPC) features supported by Guardian procedures as well as a conceptual discussion of IPC.

Passing Messages and Reply Text Between Processes

Typically, when one process sends a message to another process, the recipient process sends some data back to the sender in a reply. This model of interprocess communication is sometimes called two-way communication. Here, the requester process sends a message to the server and then typically waits until the server reads the message, processes the message, and sends a reply.

Either process can use nowait I/O or waited I/O. The example given later in this subsection uses waited I/O. The application of nowait I/O to interprocess communication, however, is no different than for any other use of nowait I/O. Refer to Handling Interprocess Messages earlier in this section.

Before sending and receiving messages, all processes involved must have initialized file FCBs for each file accessed by SIO procedures and must also have initialized the common FCB. Refer to Initializing SIO Files Using TAL or pTAL DEFINEs, earlier in this section. Specifically, the requester process must initialize at least an FCB for the server process file, and the server process file must initialize at least an FCB for the $RECEIVE file.

Sending a Message to Another Process: Reply Data Expected

To send a message to another process and receive reply data, the requester issues a READ^FILE procedure call. This call uses the same buffer to send a message and receive the reply.

The following example sends 40 bytes from the I/O buffer to the server process (designated as the process's OUT file) and waits for a reply. The reply returns in the same buffer, and the count of bytes is returned in the third parameter:

```
CALL READ^FILE(OUTFILE,BUFFER,BYTES^READ,40,40);
```

Receiving and Replying to Messages

The server process receives the message from the requester process by issuing a READ^FILE procedure call against its $RECEIVE file. If waited I/O is used, the server process waits on this call for requests to come in from a requester:

```
CALL READ^FILE(RECEIVE^FILEFCB,
               BUFFER,
               BYTES^READ);
```

When the READ^FILE call finishes, the server processes the received message before replying to the requester. The reply is done by issuing a WRITE^FILE procedure call against the $RECEIVE file:

```
CALL WRITE^FILE(RECEIVE^FILEFCB,
                BUFFER,
                WCOUNT,
                ERROR);
```

By writing a message to $RECEIVE, the server process completes the READ^FILE call issued by the requester. In addition to returning reply data and the length of the reply, the WRITE^FILE procedure can also return an error indication to the server in the *reply-error-code* parameter (the fourth parameter). You can use this parameter to pass back to the requester an application-dependent error code. The error number is picked up by the corresponding READ^FILE procedure in the requester process in the error return.

Sending and Receiving Messages in Two-Way Communication: An Example

Figure 15-5 shows an example of two-way communication between a requester process and a server process. These processes pass data between terminal users in both directions.

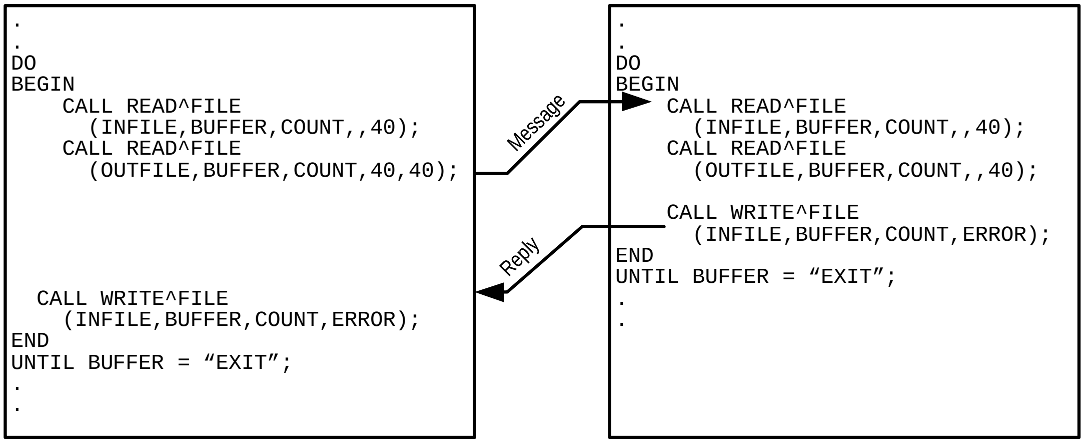
Each process reads some data from its input file and then sends the data to its output file and waits for a response. On receiving the response, the server process sends the response to its input file.

At run time, the requester process is assigned the server process as its output file; its input file defaults to the home terminal. The server process takes $RECEIVE as its input file and defaults its output file to its home terminal.

Data exchange between the requester and server processes takes place as follows:

1. The requester process prompts its terminal user for some data by issuing a READ^FILE procedure call. The user’s response is returned in the read buffer.
2. The requester sends the user’s message to the server by issuing a READ^FILE procedure call against the $SERV file. This call also waits for a response from the server.
3. Meanwhile, the server has been waiting on the READ^FILE call for a message on $RECEIVE. It now receives the message from the requester.
4. The server processes the message by sending it to its home terminal using a READ^FILE procedure call. The READ^FILE procedure waits for a response from the terminal user.
5. The server process sends the reply back to the requester by issuing a WRITE^FILE procedure call against the $RECEIVE file.

Figure 15-5. Requester and Server Processes in Two-Way Communication



```
.
.
DO
BEGIN
    CALL READ^FILE
      (INFILE,BUFFER,COUNT,,40);
    CALL READ^FILE
      (OUTFILE,BUFFER,COUNT,40,40);



  CALL WRITE^FILE
    (INFILE,BUFFER,COUNT,ERROR);
END
UNTIL BUFFER = “EXIT”;
.
.
```

```
.
.
DO
BEGIN
    CALL READ^FILE
      (INFILE,BUFFER,COUNT,,40);
    CALL READ^FILE
      (OUTFILE,BUFFER,COUNT,,40);

    CALL WRITE^FILE
      (INFILE,BUFFER,COUNT,ERROR);
END
UNTIL BUFFER = “EXIT”;
.
.
```

VST076.VSD

Passing Messages Between Processes: No Reply Data

If no data is expected in the reply from the server process, then the requester process can send messages to the server by issuing WRITE^FILE calls instead of READ^FILE. The reply is received as soon as the server reads the message, but it contains no data. There is no need for the server to explicitly send a reply. The only acknowledgment that the sender (requester) receives is to be assured that the recipient (server) read the message. This design is sometimes called one-way communication.

As with two-way communication, you must initialize all FCBs and the common FCB in both processes before trying to send messages between the processes. Refer to Initializing SIO Files Using TAL or pTAL DEFINEs earlier in this section.

Sending a Message to Another Process: No Reply Data

With all FCBs initialized, you need to do the following to send a message to another process:

- Open the server process using the OPEN^FILE procedure. You can use waited or nowait I/O. The following example uses waited I/O:

  ```
  CALL OPEN^FILE(COMMON^FCB,
                 SERVER^FILE);
  ```

- Write the message to the server process using the WRITE^FILE procedure. This example shows writing to a server opened with waited I/O:

  ```
  BUFFER ':=' "Hello Server!" -> @S^PTR;
  CALL WRITE^FILE(SERVER^FILE,
                  BUFFER,
                  @S^PTR '-' @SBUFFER);
  ```

  Here, the WRITE^FILE procedure returns when the server process has read the message from its $RECEIVE file.

Receiving Messages From Another Process: No Reply Data

Assuming that the server process has initialized a file FCB for each file that it will access, you need to do the following to receive messages from a requester process:

- Open the $RECEIVE file. You can open this file for waited or nowait I/O. The following example opens $RECEIVE for waited I/O:

  ```
  CALL OPEN^FILE(COMMON^FCB,
                 RECV^FCB);
  ```

- Read each message from $RECEIVE with a READ^FILE procedure call. Following the read operation, the read buffer contains the message and the *`count-returned`* parameter indicates the number of bytes read.

The following example reads a message from $RECEIVE:

```
CALL READ^FILE(RECV^FCB,
               READ^BUFFER,
               BYTES^READ);
```

If you choose not to explicitly reply to the message, then on the next READ^FILE call issued by the server against $RECEIVE, SIO will notice that the preceding READ^FILE was never matched with a WRITE^FILE. SIO will then send a reply for the preceding READ^FILE.

Sending and Receiving Messages in One-Way Communication: An Example

The example shown in Figure 15-6 shows a client and a server process engaging in one-way communication to pass messages from one terminal user to another. Note that the code for each process is identical, but the input and output files assigned to each process at run time are different.

Each process initializes an FCB for the input file and an FCB for the output file and opens both files with read/write access. Each process then enters a loop that reads a message from the input file and then copies the same message to the output file.

The differences between the two processes are in the input and output files that are set up at run time. For the server process, the RUN command names the server $SERV and designates $RECEIVE as its input file. The output file defaults to the home terminal as specified in the Startup message.

The requester process takes its home terminal as its input file and $SERV as its output file.

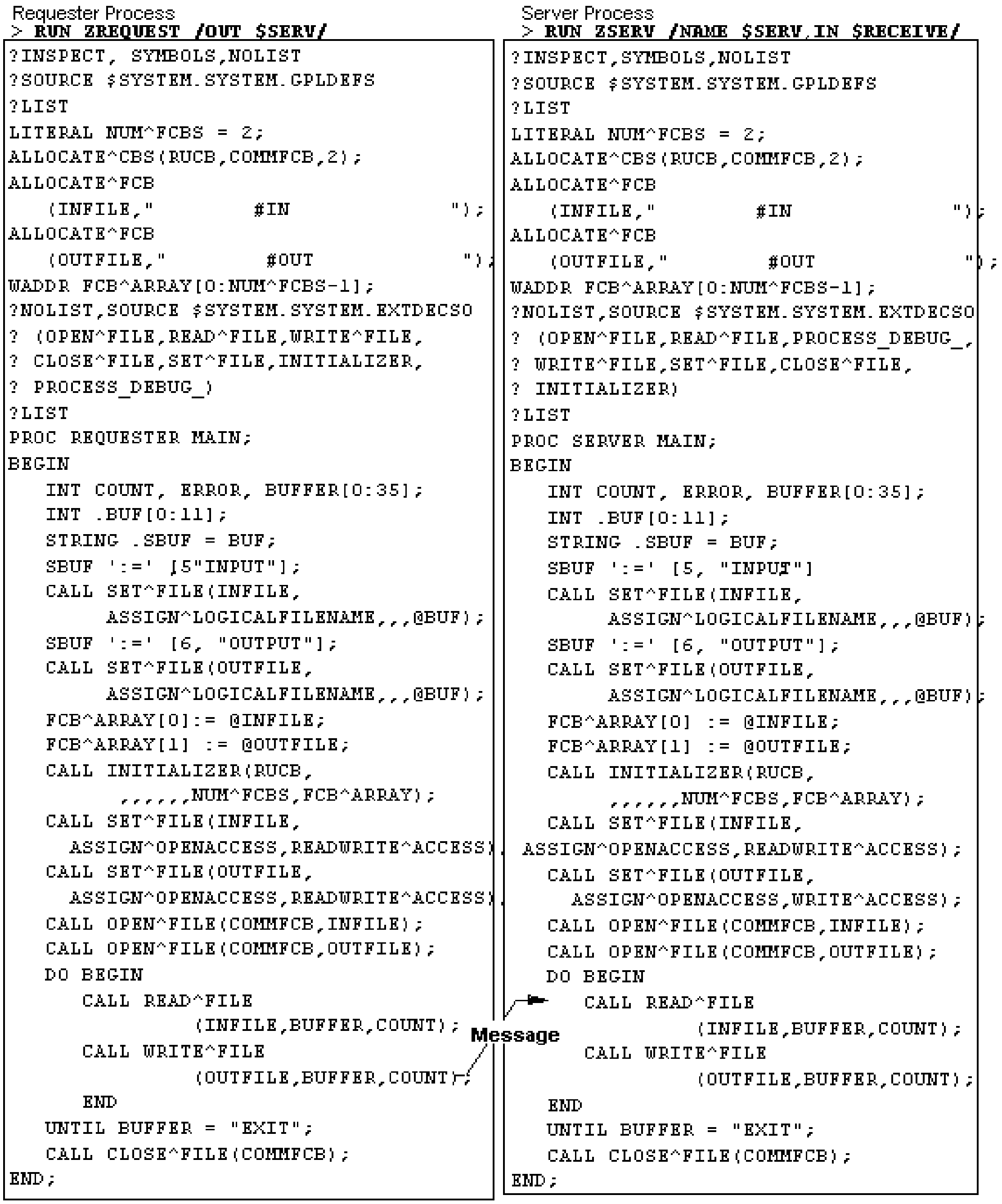
The combined action of the requester and server processes is as follows:

1. The requester process prompts the terminal user for a message using a call to the READ^FILE procedure.
2. The requester sends the same message to the server process by issuing a WRITE^FILE procedure call to $SERV. The requester then waits for the write to finish.
3. The server process reads the message from its $RECEIVE file using the READ^FILE procedure. The requester remains suspended while the server processes the request.
4. The server writes the received message to its home terminal by issuing a WRITE^FILE procedure call.
5. The requester is reactivated when the server issues the next READ^FILE call.

Both processes terminate when the user at the sending terminal types “EXIT.”

Figure 15-6 shows the code for the pTAL version of the program, and Figure 15-7 shows the code for the TAL version.

Figure 15-6. Requester and Server Processes in One-Way Communication (pTAL Version)



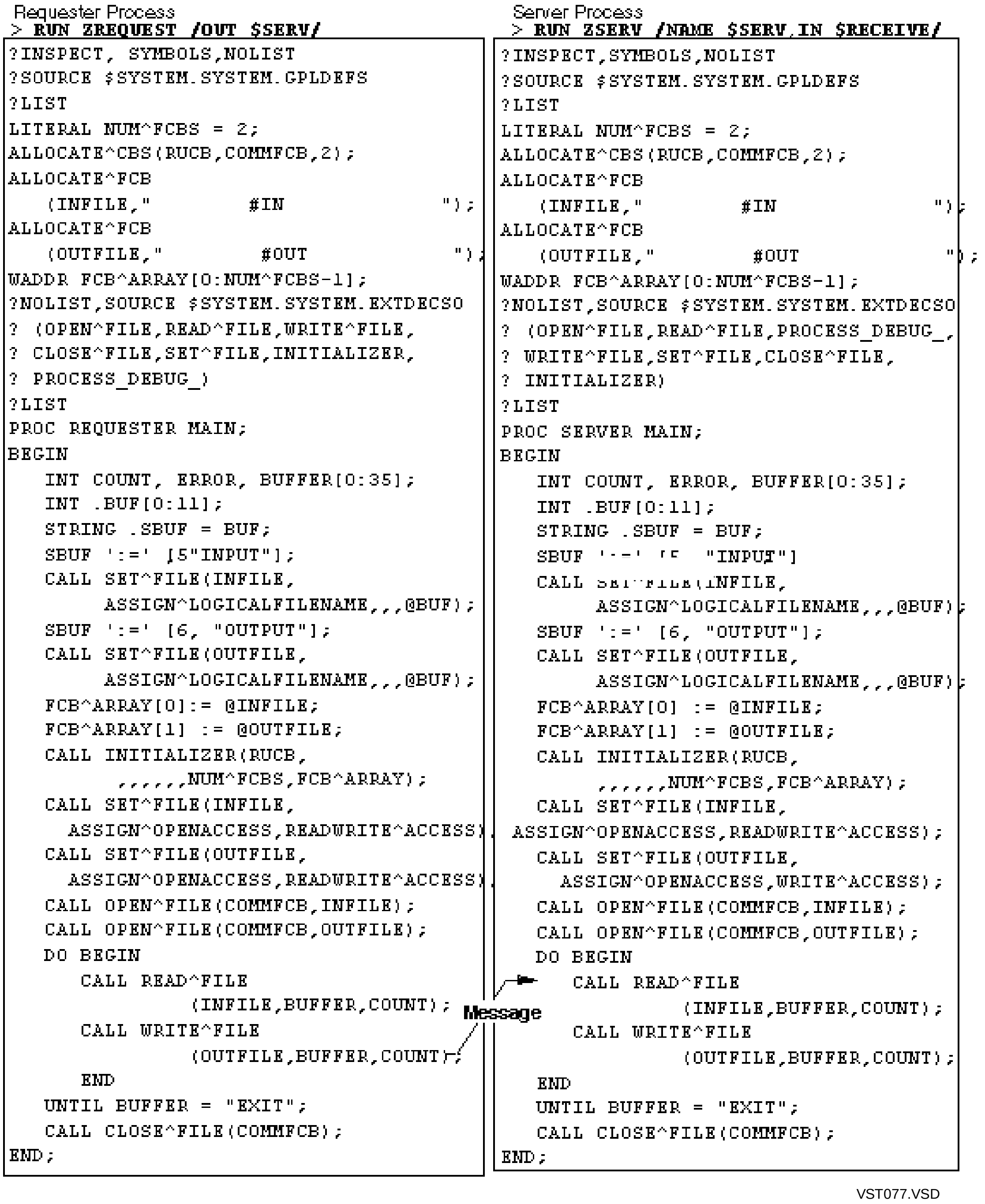
Requester Process

```
> RUN ZREQUEST /OUT $SERV/
?INSPECT, SYMBOLS,NOLIST
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
LITERAL NUM^FCBS = 2;
ALLOCATE^CBS(RUCB,COMMFCB,2);
ALLOCATE^FCB
   (INFILE,"         #IN              ");
ALLOCATE^FCB
   (OUTFILE,"         #OUT             ");
WADDR FCB^ARRAY[0:NUM^FCBS-1];
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0
? (OPEN^FILE,READ^FILE,WRITE^FILE,
? CLOSE^FILE,SET^FILE,INITIALIZER,
? PROCESS_DEBUG_)
?LIST
PROC REQUESTER MAIN;
BEGIN
   INT COUNT, ERROR, BUFFER[0:35];
   INT .BUF[0:11];
   STRING .SBUF = BUF;
   SBUF ':=' [5"INPUT"];
   CALL SET^FILE(INFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   SBUF ':=' [6, "OUTPUT"];
   CALL SET^FILE(OUTFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   FCB^ARRAY[0]:= @INFILE;
   FCB^ARRAY[1] := @OUTFILE;
   CALL INITIALIZER(RUCB,
         ,,,,,,NUM^FCBS,FCB^ARRAY);
   CALL SET^FILE(INFILE,
     ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL SET^FILE(OUTFILE,
     ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL OPEN^FILE(COMMFCB,INFILE);
   CALL OPEN^FILE(COMMFCB,OUTFILE);
   DO BEGIN
      CALL READ^FILE
               (INFILE,BUFFER,COUNT);
      CALL WRITE^FILE
               (OUTFILE,BUFFER,COUNT);
      END
   UNTIL BUFFER = "EXIT";
   CALL CLOSE^FILE(COMMFCB);
END;
```

Server Process

```
> RUN ZSERV /NAME $SERV,IN $RECEIVE/
?INSPECT,SYMBOLS,NOLIST
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
LITERAL NUM^FCBS = 2;
ALLOCATE^CBS(RUCB,COMMFCB,2);
ALLOCATE^FCB
   (INFILE,"         #IN              ");
ALLOCATE^FCB
   (OUTFILE,"         #OUT             ");
WADDR FCB^ARRAY[0:NUM^FCBS-1];
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0
? (OPEN^FILE,READ^FILE,PROCESS_DEBUG_,
? WRITE^FILE,SET^FILE,CLOSE^FILE,
? INITIALIZER)
?LIST
PROC SERVER MAIN;
BEGIN
   INT COUNT, ERROR, BUFFER[0:35];
   INT .BUF[0:11];
   STRING .SBUF = BUF;
   SBUF ':=' [5, "INPUT"]
   CALL SET^FILE(INFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   SBUF ':=' [6, "OUTPUT"];
   CALL SET^FILE(OUTFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   FCB^ARRAY[0] := @INFILE;
   FCB^ARRAY[1] := @OUTFILE;
   CALL INITIALIZER(RUCB,
        ,,,,,,NUM^FCBS,FCB^ARRAY);
   CALL SET^FILE(INFILE,
 ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL SET^FILE(OUTFILE,
     ASSIGN^OPENACCESS,WRITE^ACCESS);
   CALL OPEN^FILE(COMMFCB,INFILE);
   CALL OPEN^FILE(COMMFCB,OUTFILE);
   DO BEGIN
      CALL READ^FILE
               (INFILE,BUFFER,COUNT);
      CALL WRITE^FILE
               (OUTFILE,BUFFER,COUNT);
   END
   UNTIL BUFFER = "EXIT";
   CALL CLOSE^FILE(COMMFCB);
END;
```

Figure 15-7. Requester and Server Processes in One-Way Communication (TAL Version)



Requester Process

```
> RUN ZREQUEST /OUT $SERV/
```

```
?INSPECT, SYMBOLS,NOLIST
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
LITERAL NUM^FCBS = 2;
ALLOCATE^CBS(RUCB,COMMFCB,2);
ALLOCATE^FCB
   (INFILE,"        #IN              ");
ALLOCATE^FCB
   (OUTFILE,"        #OUT             ");
WADDR FCB^ARRAY[0:NUM^FCBS-1];
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0
? (OPEN^FILE,READ^FILE,WRITE^FILE,
? CLOSE^FILE,SET^FILE,INITIALIZER,
? PROCESS_DEBUG_)
?LIST
PROC REQUESTER MAIN;
BEGIN
   INT COUNT, ERROR, BUFFER[0:35];
   INT .BUF[0:11];
   STRING .SBUF = BUF;
   SBUF ':=' [5"INPUT"];
   CALL SET^FILE(INFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   SBUF ':=' [6, "OUTPUT"];
   CALL SET^FILE(OUTFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   FCB^ARRAY[0]:= @INFILE;
   FCB^ARRAY[1] := @OUTFILE;
   CALL INITIALIZER(RUCB,
         ,,,,,,NUM^FCBS,FCB^ARRAY);
   CALL SET^FILE(INFILE,
     ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL SET^FILE(OUTFILE,
     ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL OPEN^FILE(COMMFCB,INFILE);
   CALL OPEN^FILE(COMMFCB,OUTFILE);
   DO BEGIN
      CALL READ^FILE
               (INFILE,BUFFER,COUNT);
      CALL WRITE^FILE
               (OUTFILE,BUFFER,COUNT);
      END
   UNTIL BUFFER = "EXIT";
   CALL CLOSE^FILE(COMMFCB);
END;
```

Server Process

```
> RUN ZSERV /NAME $SERV,IN $RECEIVE/
```

```
?INSPECT,SYMBOLS,NOLIST
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
LITERAL NUM^FCBS = 2;
ALLOCATE^CBS(RUCB,COMMFCB,2);
ALLOCATE^FCB
   (INFILE,"        #IN              ");
ALLOCATE^FCB
   (OUTFILE,"        #OUT             ");
WADDR FCB^ARRAY[0:NUM^FCBS-1];
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0
? (OPEN^FILE,READ^FILE,PROCESS_DEBUG_,
? WRITE^FILE,SET^FILE,CLOSE^FILE,
? INITIALIZER)
?LIST
PROC SERVER MAIN;
BEGIN
   INT COUNT, ERROR, BUFFER[0:35];
   INT .BUF[0:11];
   STRING .SBUF = BUF;
   SBUF ':=' [5  "INPUT"]
   CALL SET^FILE(INFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   SBUF ':=' [6, "OUTPUT"];
   CALL SET^FILE(OUTFILE,
        ASSIGN^LOGICALFILENAME,,,@BUF);
   FCB^ARRAY[0] := @INFILE;
   FCB^ARRAY[1] := @OUTFILE;
   CALL INITIALIZER(RUCB,
        ,,,,,,NUM^FCBS,FCB^ARRAY);
   CALL SET^FILE(INFILE,
 ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL SET^FILE(OUTFILE,
     ASSIGN^OPENACCESS,WRITE^ACCESS);
   CALL OPEN^FILE(COMMFCB,INFILE);
   CALL OPEN^FILE(COMMFCB,OUTFILE);
   DO BEGIN
      CALL READ^FILE
               (INFILE,BUFFER,COUNT);
      CALL WRITE^FILE
               (OUTFILE,BUFFER,COUNT);
   END
   UNTIL BUFFER = "EXIT";
   CALL CLOSE^FILE(COMMFCB);
END;
```

Communicating With Multiple Processes

A server process can respond to messages from more than one requester process using either the one-way or the two-way communication model. Normally, however, SIO permits only one process pair to have the server process open at a time. To allow multiple openers, you need to handle the Open and Close messages in your application rather than let SIO handle them. This way, you can permit as many openers as you wish. See Handling System Messages, later in this section, for details.

Because your server program needs to handle Open and Close messages itself, you also need to inform SIO that the server has been opened. Otherwise, SIO will assume that the server has not been opened and will reject messages received on $RECEIVE by returning error 60. A procedure call such as the following prevents SIO from rejecting messages:

```
CALL SET^FILE(RECV^FCB,SET^RCVOPENCNT,1);
```

When multiple openers are permitted, user messages are queued on $RECEIVE for the server to read. The server simply reads the first message on $RECEIVE, processes it, then reads the next message, and so on.

Note that servers that use SIO procedures cannot queue messages for multithreaded processing like regular Guardian procedure calls can. This is because the OPEN^FILE procedure always opens $RECEIVE with a receive depth of one; therefore, the server process can work with only one message at a time. Refer to Section 6, Communicating With Processes, for details on how to perform message queuing with Guardian procedure calls.

Handling System Messages

SIO handles system messages automatically. However, you can use the SET^FILE SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY operation to select messages to be returned to your program for processing. How your program should handle system messages is described later in this subsection. First, this subsection discusses the way SIO processes system messages.

SIO processes system messages to achieve the following:

- Keep track of openers
- Keep track of the BREAK key

When keeping track of openers, SIO maintains a list of openers and limits the number of openers to one process pair. If more processes attempt to open your process, the SIO procedures reject the extra open attempts. To control attempts to open a process, SIO monitors the following system messages:

-2 (Processor down)	If the list of openers includes a process from the failing CPU, then SIO removes that process from the list.
-100 (Remote processor down)	If the list of openers includes a process from the remote CPU that failed, then SIO removes that process from the list.
-103 (Process open)	SIO checks a list of openers to see whether another process already has the current process open. If no other process has the current process open, SIO puts the process handle of the opener into the list of openers and the open continues. If the current process is already open, SIO rejects the open request with an error code of 12 (file in use).
-104 (Process close)	SIO removes the closing process from the list of openers.
-110 (Loss of communication with a network node)	If the list of openers includes a process from a CPU on the failing network node, then SIO removes that process from the list.

If you need to allow more than one opener, your process must handle the above messages.

To perform BREAK handling, SIO monitors system message -105 (or -20), the Break-on-device message. This message signals the fact that the BREAK key was pressed. SIO saves this information for processing by the CHECK^BREAK procedure. See Handling BREAK Ownership, later in this section, for details.

The remainder of this subsection shows how to override automatic system-message handling and process messages within your program. Specifically it shows how to perform the following operations:

- Mask system messages that you want your program to accept
- Read system messages from the $RECEIVE file

Selecting or Masking System Messages

Use the SET^SYSTEMMESSAGESMANY operation of the SET^FILE procedure to select which system messages you want your process to receive.

You specify a four-word mask with the SET^SYSTEMMESSAGESMANY operation. Each system message that could be sent to the process has a bit position in the mask.

If the bit is set to 1, then your process receives the corresponding system message. If the bit is 0, then your process does not receive the message.

The following example accepts only messages that indicate an attempted open or close of the current process. Bit 14 in the second word of the mask corresponds to the Open message. The example shows both the native and TNS forms of the SET^FILE call.

```
MASK[0] := %B0010000000000000;
MASK[1] := %B0000000000000011;
MASK[2] := 0;
MASK[3] := %B0000101000000000;
?IF PTAL            !Begin pTAL statement
CALL SET^FILE(RECV^FILEFCB,
              SET^SYSTEMMESSAGESMANY,,,@MASK);
?ENDIF PTAL         !End pTAL statement
?IFNOT PTAL         !Begin TAL statement
CALL SET^FILE(RECV^SYSTEMMESSAGESMANY,@MASK)
?ENDIF PTAL         !End TAL statement
```

Specifically, the process receives the following system messages:

Word	Bit	Message
0	2	(-2) Processor down
1	14	(-103 or -30) Open
1	15	(-104 or -31) Close
3	4	(-110) Loss of communication with network node
3	6	(-100) Remote Processor down

Refer to the *Guardian Procedure Calls Reference Manual* for a complete list of all system messages and mask-bit positions.

Reading System Messages

Use the READ^FILE procedure to read a system message from the $RECEIVE file. If the message you read is a system message, then the READ^FILE procedure returns error number 6, assuming you requested to receive this message using the SET^FILE SET^SYSTEMMESSAGES[MANY] operation:

```
ERROR := READ^FILE(RECEIVE^FILE^FCB,BUFFER,BYTES^READ);
IF ERROR = 6 THEN
BEGIN
   !Process system message
END;
```

Handling BREAK Ownership

This subsection describes how to use the CHECK^BREAK, GIVE^BREAK, and TAKE^BREAK procedures to manipulate BREAK ownership and respond to the pressing of the BREAK key.

This subsection is concerned with handling the BREAK key on a terminal that is initialized and opened as an SIO file. For details on how to handle the BREAK key for terminal files opened with regular Guardian procedure calls, refer to Section 10, Communicating With Terminals.

Briefly, the purpose of the BREAK key is to allow the program user to signal the process. A common example of the use of the BREAK key is to signal the TACL process to return to command-input mode when running an application. If your program does its own BREAK handling, however, you can cause the BREAK key to initiate some function of your choice.

The major BREAK-handling operations with SIO procedures are outlined below and described in detail in the paragraphs that follow.

- Take BREAK ownership using the TAKE^BREAK procedure. The BREAK function can be owned by only one process at a time. If you do not take BREAK ownership, then the effect of pressing the BREAK key is determined by the process that now owns the BREAK key.
- Check for the pressing of the BREAK key using the CHECK^BREAK procedure.
- Return BREAK ownership to the previous owner using the GIVE^BREAK procedure. You do this when your program no longer needs to respond to the BREAK key.

Using the SIO procedures to process the BREAK key is much simpler than using regular Guardian procedures, because SIO hides the complexity of handling the $RECEIVE file and checking for BREAK messages. However, use of the SIO procedures does limit BREAK handling to one terminal per process and does not support BREAK mode. If you need to handle the BREAK key across multiple terminals, or if you need to make use of BREAK mode in your program, then you should use the regular Guardian procedures as described in Section 10, Communicating With Terminals.

Most of this subsection assumes that the $RECEIVE file is either initialized and open as an SIO file or not open when you call CHECK^BREAK. If it is not open, then CHECK^BREAK opens the file for you and checks for the Break message. A call to GIVE^BREAK returns ownership of BREAK to the previous owner and closes $RECEIVE if it was opened by CHECK^BREAK.

However, if you prefer to work with $RECEIVE in a regular Guardian environment while handling the terminal as an SIO file, you can do so using the SET^FILE SET^BREAKHIT operation. How to do this is described at the end of this subsection.

Taking BREAK Ownership

Use the TAKE^BREAK procedure to take ownership of the BREAK key. Doing so allows the process to receive the Break message and respond to it. The previous owner of BREAK will no longer receive Break messages.

To take BREAK ownership, you must supply the FCB name for the terminal from which your program will accept Break messages:

```
CALL TAKE^BREAK(TERM^FCB);
```

The effect of using TAKE^BREAK with an SIO file is like issuing a call to SETMODE function 11 on a regular Guardian file, with parameter 2 of the SETMODE call equal to 0.

When your process reads a Break message from its $RECEIVE file, the default action is to return a carriage return/line feed combination of characters to the terminal. You can turn off this feature either by calling the SET^CRLF^BREAK operation of the SET^FILE procedure with the *`new-value`* parameter equal to zero or by turning off the CRLF^BREAK flag when calling the OPEN^FILE procedure:

```
CALL SET^FILE(TERM^FCB,
              SET^CRLF^BREAK,
              0);
```

You resume this feature by turning the SET^CRLF^BREAK flag on again:

```
CALL SET^FILE(TERM^FCB,
              SET^CRLF^BREAK,
              1);
```

Checking for a Break Message

Use the CHECK^BREAK procedure to check for a pressed BREAK key.

Once you have taken ownership of the BREAK key, your process will receive a Break message in its $RECEIVE file whenever the BREAK key is pressed on the terminal indicated in the call to the TAKE^BREAK procedure. Using CHECK^BREAK to read this message avoids having to read directly from $RECEIVE and having to check the condition code. You simply supply CHECK^BREAK with the FCB name for the terminal where you expect the BREAK key to be pressed:

```
CALL CHECK^BREAK(TERM^FCB);
```

If $RECEIVE is not open when you call CHECK^BREAK, SIO opens it for you and checks for the Break message; $RECEIVE stays open until you subsequently call GIVE^BREAK. If the file is already open by SIO, CHECK^BREAK checks for the Break message and leaves the $RECEIVE file open. The $RECEIVE file must not be open as a non-SIO file, or CHECK^BREAK returns an error indication.

Note. For CHECK^BREAK to work, SIO should handle the Break message. You should therefore **not** return the Break message to your program using the SET^FILE SETSYSTEMMESSAGES[MANY] operation.

Returning BREAK Ownership

Once you no longer need ownership of the BREAK key, you can return its ownership to the previous owner by issuing the GIVE^BREAK procedure. You simply supply the procedure with the FCB name for the terminal from which you were accepting BREAK ownership:

```
CALL GIVE^BREAK(TERM^FCB);
```

Handling BREAK Ownership: An Example

The following example responds to the BREAK key pressed at the terminal designated as the input file in the Startup message. Under normal operation, the process executes without interacting with the user. However, the process does periodically check for the BREAK key by issuing a CHECK^BREAK procedure call within the main loop of the program. If the process receives the Break message, then it calls the BREAK^PRESSED procedure. If no Break message is received, the process carries on executing.

The BREAK^PRESSED procedure first prompts the user for input. It then sends the user's response to the file designated as the output file in the Startup message. This procedure also provides the user with the opportunity to exit the program by typing "exit."

Note that this example includes both the native (pTAL) and TNS (TAL) forms of the INITIALIZER call. Conditional compilation directives are included to select the appropriate form so that the example will execute in either environment. Alternatively, the native form of the INITIALIZER call will work in a TNS procedure, but some additional setup is required. See Using the INITIALIZER Procedure, earlier in this section, for more information on calling the INITIALIZER procedure.

```
?INSPECT, SYMBOLS

!Source in the GPLDEFS file:

?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST

!Allocate the RUCB:

ALLOCATE^CBS(CONTROL^BLOCK,COMMON^FCB,2);

!Allocate FCBs for the input and output files:
ALLOCATE^FCB(INFILE,"        #IN             ");
ALLOCATE^FCB(OUTFILE,"        #OUT            ");

!Source in the system procedures including the SIO
!procedures.

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(DEBUG,INITIALIZER,
?                OPEN^FILE,READ^FILE,WRITE^FILE,TAKE^BREAK,
?                CHECK^BREAK,SET^FILE,CLOSE^FILE,
?                PROCESS_STOP_)
?LIST
?NOMAP,NOCODE
```

```
!------------------------------------------------------------
!Procedure called when CHECK^BREAK detects BREAK.  It prompts
!the user for some input and then writes the input to the
!output file.
!------------------------------------------------------------

PROC BREAK^PRESSED;
BEGIN
   INT .BUFFER[0:127],
       BYTES^READ;

   CALL READ^FILE(INFILE,BUFFER,BYTES^READ);
   CALL WRITE^FILE(OUTFILE,BUFFER,BYTES^READ);
   IF BUFFER = "exit" THEN
   BEGIN
      CALL GIVE^BREAK(INFILE);
      CALL CLOSE^FILE(COMMON^FCB);
      CALL PROCESS_STOP_;
   END;
END;


!------------------------------------------------------------
!Main procedure initializes and opens files, then loops while
!waiting for the BREAK key.  When the BREAK key is pressed,
!it calls the BREAK^PRESSED procedure.
!------------------------------------------------------------

PROC TERMS MAIN;
BEGIN
   INT STATE,I,J;

?IF PTAL     !Begin pTAL statements
   LITERAL NUM^FCBS = 2;
   WADDR FCB^ARRAY[0:NUM^FCBS-1];

!  Initialize the terminal file:

   FCB^ARRAY[0] := @INFILE;
   FCB^ARRAY[1] := @OUTFILE;
   CALL INITIALIZER(CONTROL^BLOCK,,,,,,,NUM^FCBS,FCB^ARRAY);
?ENDIF PTAL  !End pTAL statements

?IFNOT PTAL  !Begin TAL statement
   CALL INITIALIZER(CONTROL^BLOCK);
?ENDIF TAL   !End TAL statement

!  Assign read/write access to the input file.  This file
!  will be used as the terminal file:

   CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,
                 READWRITE^ACCESS);

!  Assign read/write access to the output file.  This file
!  will be used as the $RECEIVE file:

   CALL SET^FILE(OUTFILE,ASSIGN^OPENACCESS,READWRITE^ACCESS);
```

```
!  Open the input file:

   CALL OPEN^FILE(COMMON^FCB,INFILE);
!  Open the output file:

   CALL OPEN^FILE(COMMON^FCB,OUTFILE);
!  Take ownership of the BREAK key pressed at the terminal:

   CALL TAKE^BREAK(INFILE);
!  Loop indefinitely:

   WHILE 1 = 1 DO
   BEGIN

   !  Check whether the BREAK key has been pressed.  If so,
   !  call BREAK^PRESSED:

      STATE := CHECK^BREAK(INFILE);
      IF STATE = 1 THEN CALL BREAK^PRESSED;

   !  Main body of code.  Executes repeatedly until BREAK
   !  key is pressed:

      J := 0;
      WHILE J < 2000 DO
      BEGIN
         I := 0;
         WHILE I < 2000 DO
         I := I + 1;
         J := J + 1;
      END;
   END;
END;
```

Handling BREAK Ownership With $RECEIVE Handled as a Non-SIO File

So far, it has been assumed that the $RECEIVE file and the terminal file are both opened as SIO files. However, you can open the terminal file as an SIO file while having $RECEIVE accessed by regular Guardian procedure calls.

Because the terminal file is opened as an SIO file, TAKE^BREAK and GIVE^BREAK operate as already described. CHECK^BREAK, however, does not work because it needs $RECEIVE opened as an SIO file. You therefore need to check for the Break message by reading messages from $RECEIVE directly.

When your program receives a Break message, you can inform the SIO environment by using the SET^BREAKHIT operation of the SET^FILE procedure:

```
CALL SET^FILE(COMMON^FCB,
              SET^BREAKHIT);
```

The Break indication can then be picked up later by a CHECK^BREAK procedure call after you have closed $RECEIVE. CHECK^BREAK can then open $RECEIVE as an SIO file and check the indication set by the SET^FILE SET^BREAKHIT operation.

Handling SIO Errors

One of the advantages of using SIO procedures is that error handling is automatic. In addition to providing automatic retry for certain classes of errors, SIO also reports errors automatically by sending an ASCII error message to a designated error file. By default, the error file is the home terminal of the process.

SIO treats all errors as either fatal or retryable. Fatal errors always result in a message being sent to the error file (unless you suppress them as described below). Retryable errors requiring operator intervention can also cause a message to be sent to the error file.

SIO error messages give a brief description of the problem. For a complete description of each message and recommended action, refer to the *Guardian Procedure Errors and Messages Manual*.

Handling Error Messages

By default, SIO sends all error messages to the home terminal. The SIO procedures, however, do enable you to turn off error reporting and to redirect error messages to another file.

In addition to sending an error message to the error file, each SIO procedure can also return an error number to your program for specific processing.

Suppressing Error-Message Reporting

You can suppress the printing of error messages either when opening the file or by a SET^FILE operation on an open file. Each of these methods is described below.

Suppress error-message printing when opening the file by turning off the PRINT^ERR^MSG flag for the home terminal as follows:

```
FLAGS := 0;
FLAGS^MASK := PRINT^ERR^MSG;
CALL OPEN^FILE(COMMON^FCB,
               INFILE,
               !block^buffer!,
               !block^bufferlen!,
               FLAGS,
               FLAGS^MASK);
```

Achieve the same effect using the SET^FILE SET^PRINT^ERR^MSG operation with the *`new-value`* parameter set to 0:

```
NEW^VALUE := 0;
CALL SET^FILE(INFILE,
                  SET^PRINT^ERR^MSG,
                  NEW^VALUE);
```

Redirecting Error Messages

You can redirect error messages to an alternate file. Again you have two choices: you can specify redirection when you open a file or you can use SET^FILE.

Use the *`error-file-fcb`* parameter in the OPEN^FILE procedure to set the error-message file while opening a file. The following example redirects error messages to the output file:

```
CALL OPEN^FILE(COMMON^FCB,
               INPUT,
               !block^buffer!,
               !block^bufferlen!,
               !flags!,
               !flags^mask!,
               !max^record^len!,
               !prompt^char!,
               OUTPUT);
```

It does not matter which file you are opening when you redirect error messages. The last error file specified is the one used for all SIO files. If you do not supply the *`error-file-fcb`* parameter, then the error file does not change.

To use the SET^FILE procedure to set the error file, choose the SET^ERRORFILE operation. You must specify the common FCB in the first parameter and the error-file FCB in the third parameter. The following example redirects error messages to a file with FCB name EFILE:

```
CALL SET^FILE(COMMON^FCB,
              SET^ERRORFILE,
              EFILE);
```

Handling Fatal Errors

There are two classes of fatal errors: errors that occur when an SIO file is opened and errors that occur when the open file is accessed. For both classes of fatal error, you have the option of letting the process automatically terminate or allowing the process to continue in spite of the error.

Handling Open Errors

You can use the ABORT^OPENERR flag to choose whether to abort the process in response to a fatal error returned by the OPEN^FILE call.

The effects of setting or clearing the ABORT^OPENERR flag are explained below:

- Setting the ABORT^OPENERR flag causes the system to respond to fatal open errors by closing all files opened by the process, issuing an error message, and abnormally terminating the process. The following example sets the ABORT^OPENERR flag:

```
FLAGS := ABORT^OPENERR;
FLAGS^MASK := ABORT^OPENERR;
CALL OPEN^FILE(COMMON^FCB,
               INFILE,
               !block^buffer!,
               !block^bufferlen!,
               FLAGS,
               FLAGS^MASK);
```

- Clearing the ABORT^OPENERR flag allows the process to continue, in spite of fatal errors. If an error occurs, the open will not finish and an error message is sent to the error file. The following example shows how to clear the ABORT^OPENERR flag:

```
FLAGS^MASK := ABORT^OPENERR;
CALL OPEN^FILE(COMMON^FCB,
               INFILE,
               !block^buffer!,
               !block^bufferlen!,
               !flags!,
               FLAGS^MASK);
```

By default, the ABORT^OPENERR flag is set.

Handling I/O Errors

You can choose whether to terminate the process on receipt of a fatal I/O error or to continue without completing the I/O operation. The error message is written to the error file whether you choose to terminate or not.

Use the ABORT^XFERERR flag to choose whether to terminate the process in response to a fatal error returned by a READ^FILE or WRITE^FILE procedure call. The effects of setting or clearing the ABORT^XFERERR flag are as follows:

- If you set the ABORT^XFERERR flag, then a fatal I/O error causes the system to close all SIO files opened by this process and to terminate the process.
- If you clear the ABORT^XFERERR flag, then SIO routines report the error to the error file and the process continues. Your program can process the returned error number if desired.

  By default, the ABORT^XFERERR flag is set on.

You can set the ABORT^XFERERR flag when you open the file with the OPEN^FILE procedure, or you can change the setting of the flag using the SET^FILE SET^ABORT^XFERERR operation.

The following example sets the ABORT^XFERERR flag when the file is opened:

```
CALL OPEN^FILE(COMMON^FCB,
               INPUT,
               !block^buffer!,
               !block^bufferlen!,
               ABORT^XFERERR,
               ABORT^XFERERR);
 .
 .
ERROR := READ^FILE(INPUT,BUFFER,RCOUNT);
IF ERROR <> 0 THEN ...
   !Process nonfatal error.
```

The next example clears the ABORT^XFERERR flag:

```
CALL SET^FILE(INFILE,
              SET^ABORT^XFERERR,
              0);
 .
 .
ERROR := READ^FILE(INPUT,
                   BUFFER,
                   RCOUNT);
IF ERROR <> O THEN ...
   !Process any error.
```

Handling Retryable Errors

There are several classes of retryable errors:

- Errors that require operator intervention
- Errors resulting from BREAK activity
- Errors that are retried after a delay
- Errors resulting from path or device failure

Errors That Require Operator Intervention

Errors that require operator intervention include the following:

Error 100	Device not ready
Error 101	No write ring
Error 102	Paper out

For these errors, SIO sends an appropriate message to the error file. If the error file is not the operator’s console ($0), then SIO expects a reply. The reply can be “S” or “CTRL/Y” to stop the process, treating the error as fatal. Any other response causes the process to retry the operation and then continue.

If the error file is on a different system than the device causing the original error, then SIO sends an additional copy of the message to $0 on the system containing the device needing intervention.

After sending the error message to one or more destinations, SIO retries the operation that caused the error.

If the device in error is on the same system as the error file and if the error file is not $0, SIO expects the user to make the device ready before responding to the message. SIO therefore retries the operation immediately, and only once. If another of these errors occurs, SIO reissues the error message and prompts the user to reply.

If the error file is $0 or if the error message is being written to $0 on a remote node (because the error file is not on the same node as the device in error), then there is no way to reply to the message. SIO will then retry the operation every six seconds to see whether someone has made the device ready. In addition, SIO repeats the error message every minute. If the device is not ready after 10 minutes, SIO treats the error as fatal.

Errors Resulting From BREAK Activity

Errors resulting from BREAK activity are the following:

Error 110	Only BREAK mode requests accepted
Error 111	BREAK occurred on this request

SIO makes an internal call to the CHECK^BREAK procedure to determine whether the BREAK key was pressed for this device. If so, SIO ignores the error, because the calling program is expected to check for BREAK by issuing a CHECK^BREAK procedure call (see Handling BREAK Ownership, earlier in this section).

Otherwise, SIO retries the operation every two seconds. There is no limit on the number of retries, because BREAK ownership may have been taken by another process, which may return BREAK ownership at any time.

Errors resulting in BREAK activity do not generate error messages and are therefore transparent to the user and the programmer.

Errors That Are Retried After a Delay

Errors that are retried after a delay include the following:

Error 103	Disk not ready due to power failure
Error 112	READ or WRITEREAD preempted by operator message
Error 124	A line reset is in progress

For each of these errors, SIO retries the operation after a two-second delay.

Errors Resulting From Path or Device Failure

Errors that result from a path or device failure include the following:

Error 120	Data parity
Error 200	Device is owned by the other port
Error 201 through 231	Path error
Error 240	Line handler error; did not get started

For error 201 (unable to communicate over this path), SIO retries the error once. For the other errors, SIO retries the error if the type of operation and the type of device suggest that a retry is appropriate. In all cases where no retry is performed, SIO treats the error as fatal.

Closing SIO Files

Use the CLOSE^FILE procedure to close SIO files. You can close all SIO files by specifying the common FCB:

```
CALL CLOSE^FILE(COMMON^FCB);
```

Closing the file with CLOSE^FILE flushes the buffers, thereby ensuring data integrity. Terminating the process also closes any files that remain open; however, you would lose any data that remains in the block buffers.

For SIO files open for nowait I/O, CLOSE^FILE waits for the completion of any outstanding I/O operation and closes the file.

Initializing SIO Files Without TAL or pTAL DEFINEs

You can set up FCBs for SIO files without using TAL or pTAL DEFINEs. The advantage of initializing FCBs this way is that you can determine how many FCBs you need at run time and dynamically allocate as many FCBs as you need. However, you must write code to perform the initialization function; for example, if the process receives its input and output files through the Startup message, then you must write code to directly access the $RECEIVE file.

As when using TAL or pTAL DEFINEs, your program should make use of the $SYSTEM.SYSTEM.GPLDEFS file to provide literals used in initialization. The following compiler directives include the file in your code:

```
?NOLIST
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
```

The following list introduces the steps involved in initializing SIO files without using TAL or pTAL DEFINEs:

1. Allocate space for each FCB. This step is required.
2. Initialize an empty FCB for each file. This step is required.
3. Specify the physical file name for each FCB. This step is required.
4. Set up the file-access characteristics. This step is optional; if omitted, the system uses default values.

The following paragraphs provide details on how your program can perform the above steps. At the end of this subsection is a sample initialization.

Allocating FCBs

You must allocate space for each file FCB and the common FCB using a suitable declaration. If your program will read the Startup message using SIO, then you must also allocate space for an FCB for the $RECEIVE file.

Note. FCBs for native processes require more memory than those for TNS processes.

The following declarations allocate space for an input file, an output file, an additional data file, the $RECEIVE file, and the common FCB:

```
INT .INFILE[0:FCBSIZE - 1],
    .OUTFILE[0:FCBSIZE - 1],
    .DFILE[0:FCBSIZE - 1],
    .RECVFILE[0:FCBSIZE^D00 - 1],
    .COMMON^FCB[0:FCBSIZE^D00 - 1];
```

The literals FCBSIZE and FCBSIZE^D00 are defined in the GPLDEFS file. Note that $RECEIVE and the common FCB are the only FCBs for which FCBSIZE^D00 is used. FCBSIZE is preferred for the other FCBs because it uses less space.

Initializing FCBs

You now need to associate each of the FCBs just allocated with a physical file. For FCBs allocated with FCBSIZE, use the SET^FILE INIT^FILEFCB operation. For FCBs allocated with FCBSIZE^D00, use the SET^FILE INIT^FILEFCB^D00 operation:

```
CALL SET^FILE(INFILE,
              INIT^FILEFCB);

CALL SET^FILE(OUTFILE,
              INIT^FILEFCB);

CALL SET^FILE(DFILE,
              INIT^FILEFCB);

CALL SET^FILE(RECVFILE,
              INIT^FILEFCB^D00);

CALL SET^FILE(COMMON^FCB,
              INIT^FILEFCB^D00);
```

Naming FCBs

Use the SET^FILE ASSIGN^FILENAME operation to associate an FCB with a physical file. You must perform this operation once for each file that SIO procedures will access, including the $RECEIVE file.

The following example assigns a name to the input file. Both the native (pTAL) and TNS (TAL) forms of the SET^FILE call are shown.

pTAL:

```
CALL SET^FILE(INFILE,ASSIGN^FILENAME,,,@INFILENAME);
```

TAL:

```
CALL SET^FILE(INFILE,ASSIGN^FILENAME,@INFILENAME);
```

Note that the procedure call takes the address of the file name. In this case, the actual name of the input file has already been read from the Startup message and placed into a buffer named INFILENAME.

Setting Up File Access Without INITIALIZER

The following paragraphs describe how to set up the characteristics that control access to an SIO file that is initialized without the use of the INITIALIZER procedure. These paragraphs describe how to set the access mode, exclusion mode, record length, file code, extent sizes, and block length.

Typically, you set file characteristics by putting information into the FCB of the file using calls to the SET^FILE procedure. Because you are not using INITIALIZER, it is less convenient to set the file-access characteristics by issuing ASSIGN commands before running the program.

When dynamically allocating FCBs, you can set parameters only by using the SET^FILE procedure, not by using ASSIGNs. See Setting Up File Access earlier in this section for details.

Sample Initialization

The following procedure performs the same operations as the sample initialization shown earlier in this section under “Initializing SIO Files With the INITIALIZER Procedure.” This example, however, does not use INITIALIZER.

The procedure works like this: The names of the input and output files are delivered to the process through the Startup message. The procedure checks to see whether these names refer to the same process or terminal file. If so, that file is assigned read/write access. If the input and output files are different, then the input file is assigned read-only access and the output file write-only access.

Note. The following initialization procedure will execute in both the native and TNS environments. The procedure uses the CALL_SET^FILE_ADDRESS_. DEFINE to select the appropriate (pTAL or TAL) form of the SET^FILE procedure call in cases where the SET^FILE call passes an address value.

```
?INSPECT, SYMBOLS, NOCODE, NOMAP
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.SZYSTAL
?SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST

INT INTERACTIVE,
    ERROR,
   .COMMON^FCB[0:FCBSIZE^D00 - 1] := 0,
   .RCV^FILE[0:FCBSIZE^D00 - 1],
   .INFILE[0:FCBSIZE - 1],
   .OUTFILE[0:FCBSIZE - 1],
   .DFILE[0:FCBSIZE - 1],
   .BUFFER[0:99],
   .MOMSPHANDLE[0:ZSYS^VAL^PHANDLE^WLEN - 1],
   .MYPHANDLE[0:ZSYS^VAL^PHANDLE^WLEN - 1],
    DEVTYPE,
    LENGTH,
    JUNK;

    LITERAL PROCESS  = 0,
            TERMINAL = 6;

?NOLIST
?SOURCE
$SYSTEM.SYSTEM.EXTDECS0(SET^FILE,OPEN^FILE,PROCESS_GETINFO_,
?  READ^FILE,WAIT^FILE,CLOSE^FILE, DEVICEINFO,FNAMECOMPARE,
?  PROCESS_STOP_, PROCESS_GETPAIRINFO_,
?  PROCESSHANDLE_GETMINE_)
?LIST
```

```
!------------------------------------------------------------
!Procedure to initialize file control blocks for the input
!file and output file specified in the Startup message, as
!well as a separate data file.
!------------------------------------------------------------

PROC INIT;
BEGIN

!  Initialize the $RECEIVE file and common FCB:

?IF PTAL       !Begin pTAL statements
   STRING .SBUF = BUFFER;
?ENDIF PTAL    !End pTAL statements
?IFNOT PTAL    !Begin TAL statements
   STRING .SBUF := @BUFFER '<<' 1;
?ENDIF PTAL    !End TAL statements

   CALL SET^FILE(RCV^FILE,INIT^FILEFCB^D00);
   CALL SET^FILE(COMMON^FCB,INIT^FILEFCB^D00);
   SBUF ':=' "$RECEIVE                ";

!  The following statement calls a DEFINE which executes the
!  appropriate (native or TNS) form of the SET^FILE
!  call.

   CALL_SET^FILE_ADDRESS_(ERROR,RCV^FILE,ASSIGN^FILENAME,
                          @BUFFER);

   CALL SET^FILE(RCV^FILE,ASSIGN^RECORDLENGTH,132);

!  Open $RECEIVE for nowait I/O:

   CALL OPEN^FILE(COMMON^FCB,RCV^FILE,
                  !block^buffer!,
                  !block^bufferlen!,
                  NOWAIT,NOWAIT);

!  Allow Startup message from creator only:

   CALL PROCESSHANDLE_GETMINE_(MY^PHANDLE);
   ERROR := PROCESS_GETPAIRINFO_(MY^PHANDLE,
                                 !pair:maxlen!,
                                 !pair^len!,
                                 !primary^processhandle!,
                                 !backup^processhandle!,
                                 !search^index!,
                                 MOMS^PHANDLE);
   IF ERROR = 7 THEN CALL PROCESS_GETINFO_(!process^handle!,
                                           !file^name:maxln!,
                                           !file^name^len!,
                                               !priority!,
                                               MOMS^PHANDLE);

   !  The following statement calls a DEFINE which executes the
   !  appropriate (native or TNS) form of the SET^FILE
   !  call.

      CALL_SET^FILE_ADDRESS_(ERROR,RCV^FILE,OPENERSPHANDLE,
                             @MOMS^PHANDLE);

   !  Read the Startup message from the $RECEIVE file:

      DO
      BEGIN
         CALL READ^FILE(RCV^FILE,BUFFER,
                        !count^returned!,
                        !prompt^count!,
                        !max^read^count!,
                        1);

         DO ERROR := WAIT^FILE(RCV^FILE,LENGTH,6000D)
         UNTIL ERROR <> SIOERR^IORESTARTED;
      END
      UNTIL BUFFER = -1; !Startup message read

   !  Close the $RECEIVE file:

      CALL CLOSE^FILE(RCV^FILE);

   !  Check whether the process is being used interactively.
   !  That is, is the input file the same terminal file or
   !  process file as the output file?

      CALL DEVICEINFO(BUFFER[9],DEVTYPE,JUNK);
      INTERACTIVE :=
         IF (DEVTYPE.<4:9> = TERMINAL OR
         DEVTYPE.<4:9> = PROCESS) AND
         NOT FNAMECOMPARE (BUFFER[9],BUFFER[21]) THEN 1 ELSE 0;

   !  Initialize the input file:

      CALL SET^FILE(INFILE,INIT^FILEFCB);

   !  The following statement calls a DEFINE which executes the
   !  appropriate (native or TNS) form of the SET^FILE
   !  call.

      CALL_SET^FILE_ADDRESS_(ERROR,INFILE,ASSIGN^FILENAME,
                             @BUFFER[9]);

      CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,
         IF INTERACTIVE THEN READWRITE^ACCESS
         ELSE READ^ACCESS);

   !  If the process is run interactively, set the OUT file
   !  equal to the IN file:

      IF INTERACTIVE THEN
   @OUTFILE := @INFILE
   ELSE

!  If the process is not run interactively, initialize
!  the output file:

   BEGIN
      CALL SET^FILE(OUTFILE,INIT^FILEFCB);

!  The following statement calls a DEFINE which executes the
!  appropriate (native or TNS) form of the SET^FILE
!  call.

     CALL_SET^FILE_ADDRESS_(ERROR,OUTFILE,ASSIGN^FILENAME,
                            @BUFFER[21]);

     CALL SET^FILE(OUTFILE,ASSIGN^OPENACCESS,
                   WRITE^ACCESS);
   END;

!  Initialize the data file:

   CALL SET^FILE(DFILE,INIT^FILEFCB);
   CALL SET^FILE(DFILE,ASSIGN^OPENACCESS,
                 READWRITE^ACCESS);
END;


!------------------------------------------------------------
!Main procedure
!------------------------------------------------------------

PROC SIO^PROG MAIN;
BEGIN

   CALL INIT;
    .
    .

END;
```

Using the SIO Procedures: An Example

The TAL sample program given below uses SIO procedures to access data on disk. The disk file can be any type of disk file: structured, unstructured, or an EDIT file. The user of the home terminal accesses the data.

The program expects the names of the terminal and disk files to be supplied as the input and output file names in the Startup message. Moreover, the program will fail if the input file is not a terminal or the output file is not a disk file.

To run the program, use a RUN command such as the following:

```
1>  RUN OBJ^FILE/OUT DATAFILE/
```

You can create the data file using the FUP program. If the data file does not exist when you run the program, the program will create it as an EDIT file.

Once you start the program, you can perform read and write operations against the data file. The initial read operation always returns the first record in the file. Writes are always appended to the file.

The program is made up of the following procedures:

- The main procedure calls the INITIALIZE^FILES procedure and displays a menu for the user to select a function from. The program responds depending on the function selected:
  - For reading records, it calls the READ^RECORDS procedure.
  - For appending records, it calls the WRITE^RECORDS procedure.
  - For exiting the program, it calls the EXIT^PROGRAM procedure.
  - For an invalid selection, it calls the INVALID^SELECTION procedure.
- The INITIALIZE^FILES procedure is called by the main procedure when the program starts up. It uses the INITIALIZER procedure to initialize FCBs for the input and output files and checks that the input file is a terminal and that the output file is a disk file.

  This procedure opens the input file and also opens the output file. It opens the output file to obtain the file type (used by the OPEN^OUTPUT procedure) and to create the file in EDIT format if it does not exist. The procedure closes the output file before returning.
- The GET^COMMAND procedure prompts the user for the function to perform and then returns the function to the main procedure.
- The OPEN^OUTPUT procedure is called by the READ^RECORDS or WRITE^RECORDS procedure to open the output file. This procedure opens the file for reading or for writing, and in some cases it provides a block buffer.

  The OPEN^OUTPUT procedure receives the mode of access in its formal parameter and then opens the output file accordingly. Remember that to append to an SIO file, you must open the file in write-only mode. Therefore the file is opened in write-only mode for writing or read-only mode for reading. For that reason, opening the file is delayed until the user has selected a function.

  The OPEN^OUTPUT procedure provides a block buffer for relative, entry-sequenced, or key-sequenced files as well as for EDIT files. No block buffer is provided for an unstructured file.
- The READ^RECORDS procedure is called by the main program when the user selects reading from the main menu. This procedure calls the OPEN^OUTPUT procedure to open the disk file for read-only access and then reads the first record from the file and displays it on the terminal. It goes on to prompt the user to read

additional records. The procedure returns when either the user declines to read any more or the program reaches the end of the disk file.

- The WRITE^RECORDS procedure is similar to READ^RECORDS but appends records to the data file instead of reading. This procedure is called by the main program when the user selects appending from the main menu. First, it calls the OPEN^OUTPUT procedure to open the disk file in write-only mode, which forces writes to the end of the file. It goes on to prompt the user for the contents of the record to be written, and then it writes the record to the disk file and prompts the user to enter another record. The procedure returns when the user declines to enter another record.
- The EXIT^PROGRAM procedure is called by the main procedure when the user chooses to stop the program. This procedure uses the CLOSE^FILE procedure to flush the buffer and close the files before stopping the process.
- The INVALID^SELECTION procedure is called by the main procedure when the user makes an invalid selection from the menu. This procedure informs the user of the invalid selection and then returns to the main procedure to display the menu again.
- The WRITE^LINE procedure is called by several procedures to write a line to the IN file.

```
!This is a TAL example

?INSPECT, SYMBOLS, NOMAP
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST

!Allocate the RUCB and the common FCB:
ALLOCATE^CBS(RUCB,COMMON^FCB,2);

!Allocate an FCB for each SIO file:
ALLOCATE^FCB(INFILE,"        #IN             ");
ALLOCATE^FCB(OUTFILE,"        #OUT            ");

LITERAL   BUFSIZE   = 128;              !size of I/O buffer
LITERAL   OUTBLKLEN = 4096;             !size of output block
INT      .OUTBLKBUF[0:OUTBLKLEN/2 -1];  !output block buffer
INT      .BUFFER[0:BUFSIZE];            !read/write buffer
                                        ! (BUFSIZE + 1)
INT       FILE^TYPE;                    !type of disk file
STRING   .SBUFFER := @BUFFER[0] '<<' 1; !string pointer to
                                        ! read/write buffer
STRING   .S^PTR;                        !pointer to end of
                                        ! string

?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS0(SET^FILE,INITIALIZER,
?                       CHECK^FILE,DEVICEINFO,FNAMECOMPARE,
?                       READ^FILE,WRITE^FILE,CLOSE^FILE,
?                       OPEN^FILE,PROCESS_STOP_)
?LIST
```

```
!-------------------------------------------------------------
! The following DEFINEs make it easier to format and print
! messages.
!-------------------------------------------------------------

!  Initialize a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Print the line:

   DEFINE PRINT^LINE =
      CALL WRITE^LINE(BUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE PRINT^BLANK =
      CALL WRITE^LINE(BUFFER,0) #;

!  Print a string:

   DEFINE PRINT^STR(S) = BEGIN   START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;  END #;


!-------------------------------------------------------------
! Procedure to write a message on the terminal.
!-------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
INT   .BUF;
INT    LEN;
BEGIN
   CALL WRITE^FILE(INFILE,BUF,LEN);
END;
```

```
!-------------------------------------------------------------
! Procedure to prompt the user for the next function to
! perform:
!
!  "r" to read records
!  "a" to append records
!  "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT     COUNT^READ;

!  Prompt the user to read, append, or exit the program:

   PRINT^STR("Type 'r' for Read Log, ");
   PRINT^STR("Type 'a' for Append to Log, ");
   PRINT^STR("Type 'x' for Exit. ");
   PRINT^BLANK;

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL READ^FILE(INFILE,BUFFER,COUNT^READ,
                  @S^PTR '-' @SBUFFER);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure to open the input and output files.  The
! procedure opens structured disk files with a block buffer.
! Unstructured disk files are given no block buffer. The
! output file is opened first with write-only access to
! create it if it does not already exist. The procedure
! closes the file then reopens it with read/write access.
!------------------------------------------------------------

PROC OPEN^OUTPUT(ACCESS^MODE);
INT ACCESS^MODE;
BEGIN

!  Set the access mode of the output file to read-only for
!  read access or write-only for write access:

   CALL SET^FILE(OUTFILE,ASSIGN^OPENACCESS,ACCESS^MODE);

!  For a structured file, reopen the file with a block
!  buffer:

   IF FILE^TYPE.<0:3> = 1
      OR FILE^TYPE.<0:3> = 2
      OR FILE^TYPE.<0:3> = 3
      OR FILE^TYPE.<0:3> = 4
      THEN CALL OPEN^FILE(COMMON^FCB,OUTFILE,OUTBLKBUF,
                          OUTBLKLEN);

!  For an unstructured or odd unstructured file, do not use
!  a block buffer:

   IF FILE^TYPE.<0:3> = 0
      OR FILE^TYPE.<0:3> = 8
   THEN CALL OPEN^FILE(COMMON^FCB,OUTFILE);
END;
```

```
!------------------------------------------------------------
! Procedure to read records from the data file and display
! them on the terminal. Additional records are prompted for.
!------------------------------------------------------------

PROC READ^RECORDS;
BEGIN
   INT ERROR;
   INT COUNT^READ;

!  Open the output file for reading:

   CALL OPEN^OUTPUT(READ^ACCESS);

!  Loop until user declines to read another record:

   DO
   BEGIN

      PRINT^BLANK;

   !  Read the next record from the disk file:

      ERROR := READ^FILE(OUTFILE,BUFFER,COUNT^READ,
                         !prompt^count!,
                         BUFSIZE);
      IF ERROR = 1 THEN
      BEGIN

      !  Inform user of end-of-file and then return:

         SBUFFER ':='
                 "There are no more records in this file. "
                 -> @S^PTR;
         CALL WRITE^FILE(INFILE,BUFFER,@S^PTR '-' @SBUFFER);
         CALL CLOSE^FILE(OUTFILE);
         RETURN;
      END

      ELSE
      BEGIN

      !  Write the record to the terminal:

         CALL WRITE^FILE(INFILE,BUFFER,COUNT^READ);
      !  Prompt the user to read another record:

         PRINT^BLANK;
         SBUFFER ':='
                 "Do You Wish to Read Another Record (y/n)? "
                 -> @S^PTR;
         CALL READ^FILE(INFILE,BUFFER,COUNT^READ,
                        @S^PTR '-' @SBUFFER,BUFSIZE);
      END;
   END
   UNTIL NOT (SBUFFER = "y" OR SBUFFER = "Y");

!  Close the output file:

   CALL CLOSE^FILE(OUTFILE);
END;
```

```
!------------------------------------------------------------
! Procedure to write a new record to the disk file.
!------------------------------------------------------------

PROC WRITE^RECORDS;
BEGIN
   INT READ^OR^WRITE;
   INT COUNT^READ;

!  Open the output file for writing:

   CALL OPEN^OUTPUT(WRITE^ACCESS);

   DO BEGIN

      PRINT^BLANK;

   !  Prompt the user to enter a record:

      SBUFFER ':=' "Type the New Record:  " -> @S^PTR;
      CALL READ^FILE(INFILE,BUFFER,COUNT^READ,
                     @S^PTR '-' @SBUFFER,BUFSIZE);

   !  Write the record to the disk file:

      CALL WRITE^FILE(OUTFILE,BUFFER,COUNT^READ);

   !  Prompt the user to write another record:

      PRINT^BLANK;
      SBUFFER ':='
              "Do You Wish to Write Another Record (y/n)?  "
              -> @S^PTR;
      CALL READ^FILE(INFILE,BUFFER,COUNT^READ,
                     @S^PTR '-' @SBUFFER,BUFSIZE);
   END
   UNTIL NOT (SBUFFER[0] = "y" OR SBUFFER[0] = "Y");

!  Close the output file:

   CALL CLOSE^FILE(OUTFILE);
END;
```

```
!------------------------------------------------------------
! Procedure to exit the program.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN

!  Close all SIO files:

   CALL CLOSE^FILE(COMMON^FCB);

!  Stop the program:

   CALL PROCESS_STOP_;
END;

!------------------------------------------------------------
! Procedure to respond to an invalid function. Any function
! other than "a," "r," or "x" calls this procedure.
!------------------------------------------------------------

PROC INVALID^SELECTION;
BEGIN

   PRINT^BLANK;

!  Inform user of invalid selection and then return:

   PUT^STR
     ("Invalid Selection, you must type 'r,' 'a,' or 'x.' ");
END;

!------------------------------------------------------------
! Procedure for initializing all SIO files used by this
! application.
!------------------------------------------------------------

PROC INITIALIZE^FILES;
BEGIN

LITERAL  DISK     = 3;          !identify process file
LITERAL  TERMINAL = 6;          !identify terminal file
LITERAL  ABEND    = 1;          !to send Abend message on
                                ! PROCESS_STOP_
INT      DEVICE^TYPE;           !type of device
INT      PHYS^REC^LEN;          !length of physical record
INT      L^INFO[0:9];           !for device information
INT     .INFNAME,               !input file name
        .OUTFNAME;              !output file name
INT     .BUF[0:11];             !contains a logical file name
STRING  .SBUF := @BUF '<<' 1;  !string pointer to BUF
```

```
!  Assign a logical file name to each SIO file:

   SBUF ':=' [5,"INPUT"];
   CALL SET^FILE(INFILE,ASSIGN^LOGICALFILENAME,@BUF);
   SBUF ':=' [6,"OUTPUT"];
   CALL SET^FILE(OUTFILE,ASSIGN^LOGICALFILENAME,@BUF);

!  Initialize the FCBs:

   CALL INITIALIZER(RUCB);

!  Get the physical file names for the input and output
!  files:

   @INFNAME := CHECK^FILE(INFILE,FILE^FILENAME^ADDR);
   @OUTFNAME := CHECK^FILE(OUTFILE,FILE^FILENAME^ADDR);

!  Make sure that the input file is a terminal:

   CALL DEVICEINFO(INFNAME,DEVICE^TYPE,PHYS^REC^LEN);
   IF (DEVICE^TYPE.<4:9> <> TERMINAL)
      THEN CALL PROCESS_STOP_(!process^handle!,
                             !specifier!,
                             ABEND);

!  Open the input file for reading and writing:

   CALL SET^FILE(INFILE,ASSIGN^OPENACCESS,READWRITE^ACCESS);
   CALL OPEN^FILE(COMMON^FCB,INFILE);

!  Make sure that the output file is a disk file:

   CALL DEVICEINFO(OUTFNAME,DEVICE^TYPE,PHYS^REC^LEN);
   IF (DEVICE^TYPE.<4:9> <> DISK) THEN
   BEGIN
      PRINT^STR ("Illegal Output File Name ");
      CALL PROCESS_STOP_(!process^handle!,
                         !specifier!,
                         ABEND);
   END;

!  Open the output file with write-only access to create it
!  if it does not exist. Also get the file type to know
!  whether to use a block buffer when we reopen the file:
   CALL SET^FILE(OUTFILE,ASSIGN^OPENACCESS,WRITE^ACCESS);
   CALL OPEN^FILE(COMMON^FCB,OUTFILE,OUTBLKBUF,OUTBLKLEN);
   FILE^TYPE := CHECK^FILE(OUTFILE,FILE^FILEINFO);
   CALL CLOSE^FILE(OUTFILE);
END;
```

```
!------------------------------------------------------------
! Main procedure prompts the user to enter a function to read
! or write to the disk file or exit the program.
!------------------------------------------------------------

PROC LOG^PROG MAIN;
BEGIN
   STRING CMD;

!  Initialize the SIO files:

   CALL INITIALIZE^FILES;

!  Loop until user requests to exit:

   WHILE 1 = 1 DO
   BEGIN

      PRINT^BLANK;

      CMD := GET^COMMAND;

   !  Call procedure depending on function selected:

      CASE CMD OF
      BEGIN

         "r" -> CALL READ^RECORDS;

         "a" -> CALL WRITE^RECORDS;

         "x" -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^SELECTION;
      END;
   END;
END;
```

16

Creating and Managing Processes

This section shows how to use Guardian procedures to manage Guardian processes. First, it describes how the operating system manages the process environment; then it goes on to discuss how to perform the following operations in an application process:

- Create new Guardian processes, including naming the new process, running the process in a waited or nowait manner, and setting the various attributes that a process can have, including whether the process runs at a high PIN or low PIN (PROCESS_LAUNCH_ and PROCESSNAME_CREATE_ procedures).
- Send the startup sequence of messages to a process.
- Monitor a child process to make sure that it is still running (CHILD_LOST_ procedure).
- Delete your own or some other process (PROCESS_STOP_ procedure) or control which processes have the authority to delete your process (SETSTOP procedure).
- Suspend and activate processes (PROCESS_SUSPEND_ and PROCESS_ACTIVATE_ procedures).
- Get and set information about specified processes (PROCESS_GETINFO[LIST]_, PROCESS_SETINFO_, and PROCESS_SETSTRINGINFO_ procedures).
- Manipulate process identifiers, including retrieving information from a process handle (PROCESSHANDLE_DECOMPOSE_ procedure) and converting between process handles and process file names (FILENAME_TO_PROCESSHANDLE_, PROCESSHANDLE_TO_FILENAME_, and PROCESSHANDLE_TO_STRING procedures).
- Control the placement of processes onto IPUs (IPUAFFINITY_GET_, IPUAFFINITY_SET_, and IPUAFFINITY_CONTROL_).

Refer to the *Guardian Procedure Calls Reference Manual* for complete details on the procedure calls indicated above.

Process Management Overview

When any program runs on the system, it is called a process. The term “program” refers to a static group of instruction codes and initialized data (like the output of a compiler); the term “process” identifies the dynamically changing states of an executing program.

The same program (whether an application or system program) can be concurrently executing several times in the same CPU or in different CPUs. Each execution is considered a separate process.

A process consists of the following:

- Code areas in virtual memory that contain the instruction codes to be executed. These code areas are shared by all processes in the same CPU that execute the same program file. The instructions in the code areas in virtual memory are derived from the code part of the program file on disk.
- Data areas in virtual memory that contain the program variables and temporary storage that is private to the process. Even if other processes use the same code areas, each process has its own private data areas. The disk part of the data area is obtained from the Kernel-Managed Swap Facility (KMSF), or, in some cases, from a designated swap file.
- A process control block (PCB) that is used by the operating system to control process execution. The PCB contains pointers to the process code and data areas, retains process context when the process is suspended, and contains pointers to files opened by the process.

In addition to a PCB associated with every process, the operating system maintains several other tables of information to keep track of processes. A collection of such tables, known as the destination control table (DCT), contains information about all named processes on the system. This table is a system-wide table and therefore remains visible even if a CPU should fail.

Process Identifiers

A D-series system has an architectural limit of 64K processes that can concurrently run on each CPU. The practical limit is significantly smaller than this number and is constrained by memory and other resources. However, because you can have up to 16 CPU modules in a system and up to 256 such systems in a network, there is the potential for many millions of processes. The operating system therefore provides the following methods of identifying processes:

- Process file names
- Process handles

Process File Names

In the operating system, most objects are considered to be files. Just like disk files and devices, a process can also be considered to be a file. The file name for a process is known as a process file name; it can be used, for example, to open a process for communication by passing it to the FILE_OPEN_ procedure.

A process descriptor is a process file name returned by a system procedure. A process descriptor is always unqualified; that is, it cannot contain process qualifiers like the named form of a process file name can.

See Section 2, Using the File System, for a complete discussion of process file names, including syntax definitions and information about how to use process file names when opening a process file.

Process Handles

A process handle can be considered to be the address at which a process resides. The process handle is 10 words long and contains the following information:

- The process identification number (PIN) which is unique among all current processes on a given CPU. PIN values range from 0 to 65534. PINs 0 through 255 are called low PINs; PINs 256 through 65534 are called high PINs. Processes on a D-series system can run at a high PIN or a low PIN. Processes on a C-series system can run only at a low PIN.

  Note that PIN 255 is a special-case low PIN that is used by high-PIN processes to communicate with an unconverted C-series process. The range of usable low PINs is therefore 0 through 254.
- The ID of the CPU on which the process runs.
- A verifier to uniquely identify a process over time.
- A process pair index that enables the operating system to find the other member of a process pair.
- A type field that identifies whether the process is named or unnamed and whether the system on which the process runs is a D-series system or a C-series system.

Like process file names, a process handle is returned by the system when you create a process. Process handles, however, are not file names; they are used to identify the process to other process-related procedure calls, such as PROCESS_ACTIVATE_ and PROCESS_SUSPEND_.

To obtain the information contained in a process handle, you can use the PROCESS_GETINFO_ procedure as described later in this section.

Programs and Processes

A program is a sequence of instructions and data that become a process when executed.

A program file is an executable object file. It contains primarily executable code, but may also contain other components such as initial or read-only data and linkage information. Unlike other object files, a program file has a main procedure.

Object files are produced by compilers that translate the source program, written in a language such as TAL or C, into object code. They are also produced by linkers, such as the Binder and `nld` utilities (on TNS/R systems) or `eld` utilities (on TNS/E systems) which link object files together. For execution, the code and some of the data in the object file are mapped into the virtual memory of the CPU.

Kinds of Programs

These kinds of programs run on native CPUs. They are:

- TNS/E native programs
- TNS/R native programs
- TNS programs
- Accelerated programs

A native process—a process that runs in native mode—consists entirely of native-compiled instructions. Those instructions are RISC instructions on a TNS/R system and Itanium instructions on a TNS/E system. A native process is initiated by executing a native program. Unlike TNS processes, native processes do not use or emulate TNS architecture-specific constructs, such as TNS registers or 16-bit addressing.

Executable code for a TNS/E native process is contained in the following objects:

- The initial program of the process, called user code. This code is read from the program file.
- Dynamic-link libraries (DLLs). These include:
    - The system library, which contains system-related procedures and operating system code that is accessible by the process using system procedure calls. The system library consists of a set of implicit DLLs.
    - Other DLLs supplied by HP, such as the C run-time library.
    - User-created DLLs.

A TNS/R native program contains TNS/R native object code. Native object code is produced by a native compiler and consists entirely of RISC instructions that have been arranged to take full advantage of the RISC architecture. Refer to the *C/C++ Programmer's Guide* and the *pTAL Programmer's Guide* for information about native compilers.

A TNS program contains TNS object code. TNS object code is produced by a compiler that is not native. TNS object code executes TNS instructions facilitated by millicode. Millicode is assembled program code, consisting of RISC instructions, that implements various TNS low-level functions on a TNS/R CPU. Actual TNS CPUs do not support D40 or later versions of the operating system.

An accelerated program contains accelerated object code. Accelerated object code is produced by the Accelerator, a program that processes a TNS object file to run more efficiently on a TNS/R CPU. An accelerated object file consists of Accelerator-generated RISC instructions as well as the original TNS instructions. For more information on using the Accelerator, see the *Accelerator Manual*.

The accelerated version of an object file almost always runs faster than the TNS version; the native version of an object file almost always runs faster than the accelerated version. The actual differences in execution speed between TNS, accelerated, and native versions of the same program may vary, depending on the constructs that are used in the source code.

Kinds of Processes

A TNS/R native process is a process that is initiated by executing a TNS/R native program. A native process executes in the native operating environment of the TNS/R CPU.

A TNS process is a process that is initiated by executing a TNS or accelerated program. A TNS process executes within an emulated TNS operating environment.

Process Organization

Executable code for a TNS/R process is contained in the following objects:

- The initial program of the process, called user code. This code is read from the program file.
- A system library containing system-related procedures. This includes operating system code that is accessible by the process using system procedure calls.
- Optionally, a user library containing code that can be shared among several applications. This code is read from an object file containing the user-supplied library.
- Optionally, for TNS/R native processes, one or more TNS/R native shared run-time libraries (SRLs). An SRL functions similar to an extension of the system library except that the System Library code is intercepted by the User Libraries. However, the shared run-time libraries (SRLs) are not intercepted by the user libraries.

Executable code for a TNS/E native process is contained in the following objects:

- The initial program of the process, called user code. This code is read from the program file.
- Dynamic-link libraries (DLLs). These include:
  - The system library, which contains system-related procedures and operating system code that is accessible by the process using system procedure calls. The system library consists of a set of implicit DLLs.
  - Other DLLs supplied by HP, such as the C run-time library.
  - User-created DLLs.

When a process is created, it occupies space in virtual memory. The following paragraphs discuss the basic organization of a process in terms of code spaces, which are associated with the objects in the preceding list, and data spaces.

Code Spaces

A process has distinct code spaces that contain executable code. For TNS processes the code spaces are designated as follows:

- UC (user code)

- UL (user library)
- SC (system code)
- SL (system library)
- SCr (system code RISC)
- SLr (system library RISC)

For TNS/R native processes the code spaces are:

- UCr (user code RISC)
- Native UL (user library)
- SRL (shared run-time libraries)
- SCr (system code RISC)
- SLr (system library RISC)
- Ordinary DLLs (G06.20 and later RVUs)

For TNS/E native processes the code spaces are:

- UC (user code)
- UL (DLL user library)
- Implicit DLLs (correspond to TNS/R SCr and SLr)
- public DLLs (correspond to public SRLs on TNS/R)
- ordinary DLLs

Many procedures in the SCr and SLr code spaces are associated with special shell procedures, known as To-RISC shells, that make it possible to call them from procedures in the TNS code spaces. Because these native procedures are accessible by procedures in the TNS code spaces, the environment of a TNS process effectively includes the SCr and SLr code spaces, in addition to the four TNS code spaces. The term “system library” can refer collectively to the SC, SL, SCr, and SLr code spaces.

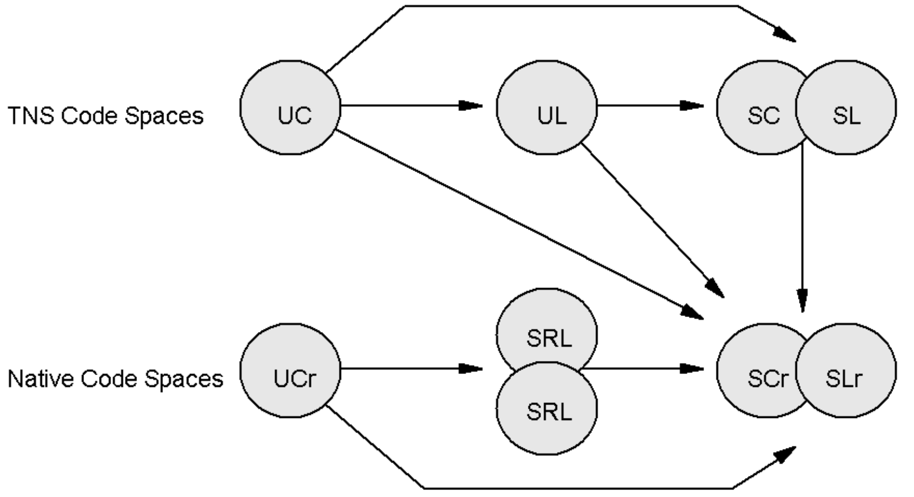
A native procedure (that is, a procedure in a native code space) cannot call a TNS procedure (that is, a procedure in a TNS code space), so the TNS code spaces are not part of a native process environment.

A procedure in a particular code space can make direct calls by name to other procedures in the same code space. It can also make calls to procedures in other code spaces as illustrated in Figure 16-1. Procedures in the user library or SRL spaces cannot ordinarily call procedures in user code. Procedures in the system code or system library spaces cannot ordinarily call procedures in the user library, SRL, or user code spaces.

A procedure that might not have direct access to a procedure in another code space can often call the other procedure indirectly by using a procedure pointer rather than a procedure name. For example, the HEAPSORTX_ procedure in the system library calls a comparison routine, typically in user code, that is passed as a procedure parameter. However, such calls are not possible between TNS and native procedures except where a TNS procedure is calling a native procedure via its To-RISC shell.

In general, variables defined in one code space are not accessible by name from other code spaces, although they can be passed as parameters. Exceptions to this are SRL and native UL instance data. SRLs and native ULs can have their own global variables that are allocated their own data spaces called instance data. Items within the instance data can be exported, making them visible to user code and to other SRLs.

Figure 16-1. TNS and Native Process Code Spaces



An arrow indicates that procedures in the code space that is pointed to can be called by name from the code space at the arrow's origin. A pair of code spaces that are represented as overlapping can call each other and can be regarded as a single entity for the purposes of this illustration.

VST142.VSD

There are two distinct sets of procedure names. All TNS and accelerated procedures exist in the TNS name set and can be called only by TNS or accelerated procedures; the Binder utility works with this set. All TNS/R native procedures exist in the native name set and can be called from native procedures; the `nld` and `noft` utilities work with this set. All TNS/E native procedures exist in the native name set and can be called from native procedures; the `eld` and `enoft` utilities work with this set.

A To-RISC shell projects a native procedure name into the TNS name set so that TNS and accelerated procedures can call it. A few system procedures, such as ARMTRAP, are in only the TNS set and cannot be called from native procedures; a few others are in only the native set and cannot be called from TNS or accelerated procedures. Most of the Guardian procedures are in both sets. Sometimes there are distinct TNS and native procedures having the same name and basic function, but more often a To-RISC shell lets the native code serve both kinds of caller.

The SYSTEMENTRYPOINT procedure accepts names from the TNS set and returns 16-bit labels for TNS procedures and To-RISC shells in the system library. The SYSTEMENTRYPOINT_RISC_ procedure accepts names from the native set and returns 32-bit addresses for native procedures in the system library. Both procedures are described in the *Guardian Procedure Calls Reference Manual*.

Data Spaces for TNS Processes

When a process is created, several data segments are allocated for its use. A TNS process has the following data segments:

- A user data segment, containing the program global data and the user data stack for TNS procedures.
- A main stack segment, containing the stack for unprivileged native procedures.
- A priv stack segment, containing the stack for privileged native procedures.
- A process file segment (PFS), used by the operating system.

The user data stack, in the TNS user data segment, is where the stack frame, or activation record, is dynamically managed for each TNS procedure that is called. This means that information, including formal parameters, a return address, and local data, is put on the stack for each TNS procedure that is called; this information is removed from the stack when the procedure finishes.

When TNS code calls a native procedure through its To-RISC shell, execution automatically switches either to the main stack, for an unprivileged native procedure, or to the priv stack, for a privileged native procedure. Execution switches back to the TNS user data stack when the native procedure finishes.

The TNS user data segment has a fixed size that you can specify, up to 128 kilobytes (KB) of virtual memory. The lower 64 KB of this space, containing program global data and the user data stack, is managed for you by the operating system. The remaining 64 KB is also available for use, but TAL and pTAL programs must manage the space themselves. The Common Run-Time Environment (CRE) manages that area for programs in other TNS languages.

If your TNS or accelerated program needs more than 128 KB of user data space, you can add extended data segments to your process. Section 17, Managing Memory, provides details on how to add segments and perform other memory-management activities.

Note. In TNS and accelerated programs, some portion of the user data stack might be used for managing data for system procedure calls. TNS and accelerated programs should allow at least 700 bytes of the user data stack for use by system procedure calls.

Data Spaces for Native Processes

A TNS/R native process has the following data segments:

- A globals-heap segment, containing program global data and, optionally, a heap.
- A main stack segment, containing the stack for unprivileged native procedures.
- A priv stack segment, containing the stack for privileged native procedures.
- Zero or more SRL data segments, used for optional global data owned by each shared run-time library (SRL).
- A process file segment (PFS), used by the operating system.

A TNS/E native process has the following data segments:

- A globals-heap segment, containing program global data and, optionally, a heap.
- A main memory stack for nonprivileged TNS/E native procedures
- A privileged memory stack for privileged procedures
- A main register stack engine (RSE) backing store for nonprivileged procedures
- A privileged RSE backing store for privileged procedures
- Zero or more DLL data segments
- Zero or more private DLL code segments
- A process file segment (PFS), used by the operating system
- Optional program-allocated extended data segments (selectable or flat segments)

For native C and C++ programs, the native Common Run-Time Environment (CRE) automatically manages a heap in the globals-heap segment. The heap is optional for other programs.

The main stack segment contains the stack for unprivileged native procedure calls. Execution automatically switches to the priv stack when a privileged procedure is called, and switches back to the main stack when that privileged procedure finishes. (An unprivileged procedure can call only selected privileged procedures, which have the CALLABLE attribute.)

RISC and Itanium stack growth is as follows: (Note that TNS stacks grow upwards.)

- RISC stacks grow downwards (from higher to lower addresses)
- The Itanium RSE backing store grows upwards
- Itanium memory stacks grow downwards

On both TNS/R and TNS/E CPUs, the main stacks (main RISC stack on TNS/R, main memory stack on TNS/E) and the heap grow automatically as needed, to a maximum size. On the TNS/R platform, the default maximum stack size is 1 MB, and on the TNS/E platform, the default maximum stack size is 2 MB. You can increase the maximum stack size via an `eld` or PROCESS_LAUNCH_ parameter up to a limit of 32MB.

Also on both platforms, the heap can grow to the maximum size of the globals-heap segment less the size of the global data. On the TNS/R platform, the maximum globals-heap size is 1.1 GB. On the TNS/E platform, the maximum globals-heap size is 1536 MB.

If your native program needs additional space for user data, you can add extended data segments to your process. Section 17, Managing Memory, provides details on how to add segments and perform other memory-management activities.

Process Security

The system provides many tools for managing processes on the system, both at the command-interpreter level and the procedure-call level. To prevent users from using these tools to interfere with another user's process (for example, to delete someone else's process) or access privileged data, the operating system provides tools for protecting processes from each other and for protecting data from indiscriminate access.

Each Guardian process is assigned a creator access ID (sometimes known as the CAID), a process access ID (or PAID), and a stop mode. The following paragraphs describe how the creator access ID, process access ID, and stop mode work together to provide process security.

Creator Access ID and Process Access ID

The creator access ID (CAID) identifies the user who initiated the creation of the process. The process access ID, which is often the same as the creator access ID, determines whether the process has the authority to make file accesses (see Section 2, Using the File System, for a discussion of file-access permissions). The process access ID is also used to determine whether restricted actions against a process (such as stopping the process or invoking the debugger) are possible.

Normally, the creator access ID and process access ID are set to the same value as the process access ID of the creating process. For example, if the TACL process with process access ID 4,56 starts a process $P1, then $P1 has creator access ID 4,56 and process access ID 4,56. Similarly, if process $P1 starts process $P2, process $P2 will have a process access ID of 4,56 and a creator access ID of 4,56. Any of these processes can then access any files belonging to user 4,56 and stop or invoke the debugger on any process started by this user.

The general rule for file access or performing any of the above actions on a process is that your process must have a process access ID equal to one of the following:

- The super ID (255, 255)
- The process access ID of the group manager of the target file or process
- The process access ID of the target process

You can set the process access ID equal to the owner ID of the object file instead of to the process access ID of the creator process. Doing so gives the new process the access permissions of the file owner instead of the creator. The owner of the object file must set up this feature, however, either by using the FUP SECURE PROGID command or programmatically by using a call to the SETMODE procedure.

An example of the use of this feature might be in setting up a password file. Each user needs to have the ability to set a password, but the password file and the program file containing the code that updates the password file would typically be owned by the super ID user. By securing the program file with FUP SECURE PROGID, the super ID user gives every user the ability to change a password.

Stop Mode

Normally, the user that can stop a process is the user that started the process (creator access ID), the user's group manager, or the super ID user. However, you can change this stop mode in a program using the SETSTOP procedure to enforce various levels of security against stopping your process.

Stop mode can be set to restrict the ability to stop your process to one of the following:

- Any process on the system
- A process with process access ID equal to that of the super ID user, the group manager of the target process, or the target process itself
- No process at all, except that your process can delete itself

Relationship With Other Processes

In many Guardian applications, the relationships among processes are critical to the operation of the application. For a process to manage the processes it has created, the creator process needs to be kept informed when one of its offspring is deleted. Similarly, a process that manages a batch job needs to be kept informed about the status of processes within the job.

So that process-deletion messages can be sent to the appropriate process, the system keeps track of relationships between processes. The method the system uses to keep track of these relationships depends on whether the process is named or unnamed and on whether you have a single process or two processes running as a process pair.

Figure 16-2 shows the relationships between a process and named and unnamed processes that the process has created.

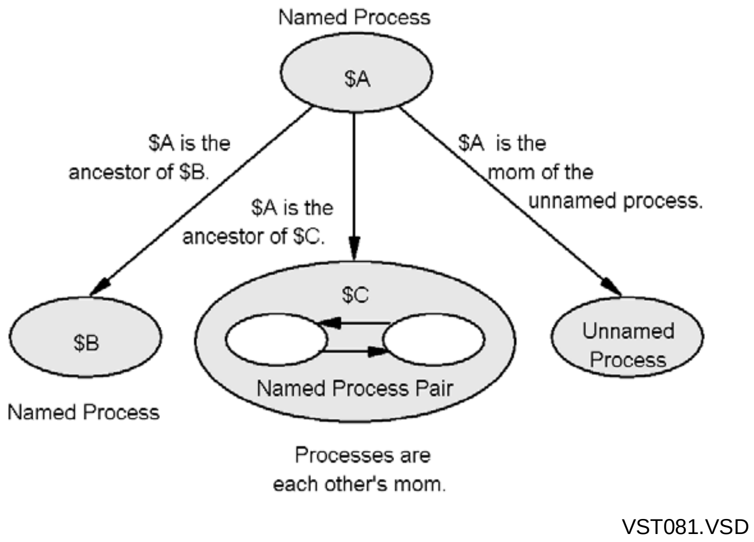
Relationship With a Named Process

Figure 16-2 shows process $A creating two named processes: a single named process $B and a named process pair $C.

Process $A is known as the **ancestor** of process $B. The relationship between $A and $B is recorded in the destination control table (DCT). If $B is deleted, then the operating system uses the DCT entry to find out where to send the process-deletion message (system message -101). Note that because the DCT is a system-wide table, the operating system can find this information even if process $B was deleted because of a CPU failure.

Process $A is also the ancestor of process $C. Because $C is a process pair, process $A gets the deletion message only if both members of the process pair are deleted. So long as either member of the process pair is running, process $A does not need to be informed.

Figure 16-2. Mom and Ancestor Processes


Named Process
$A
$A is the
ancestor of $B.
$A is the
ancestor of $C.
$A is the
mom of the
unnamed process.
$B
Named Process
$C
Named Process Pair
Processes are
each other's mom.
Unnamed
Process
VST081.VSD

Relationship With an Unnamed Process

Referring again to Figure 16-2, process $A also creates an unnamed process. For unnamed processes, the linkage to the creator process is provided in the mom field of the PCB. The mom field contains the process handle of the creator process. If the unnamed process is stopped, then the operating system uses the address in the mom field to send a process-deletion message to the creator process. In this case, the creator process is known as the mom process.

If the unnamed process is deleted because of a CPU failure, then the linkage information in the PCB goes away with the CPU. The mom process therefore does not receive the process-deletion message. It is up to the mom process to check for the CPU down message. The mom process must have issued the MONITORCPUS procedure call for the appropriate CPU, if the process will receive the CPU down message.

Relationship of Processes Within a Job

A job is a collection of related processes that are grouped for batch processing. The NetBatch utility identifies the job that a process belongs to by the job number held in the PCB of each process; processes belonging to the same job have the same job number.

A job ancestor is the process that started the first process in a job. For named and unnamed processes, when a new process is started by a process within the job, two job-related pieces of information are passed from the creator process to the new process and saved in its PCB:

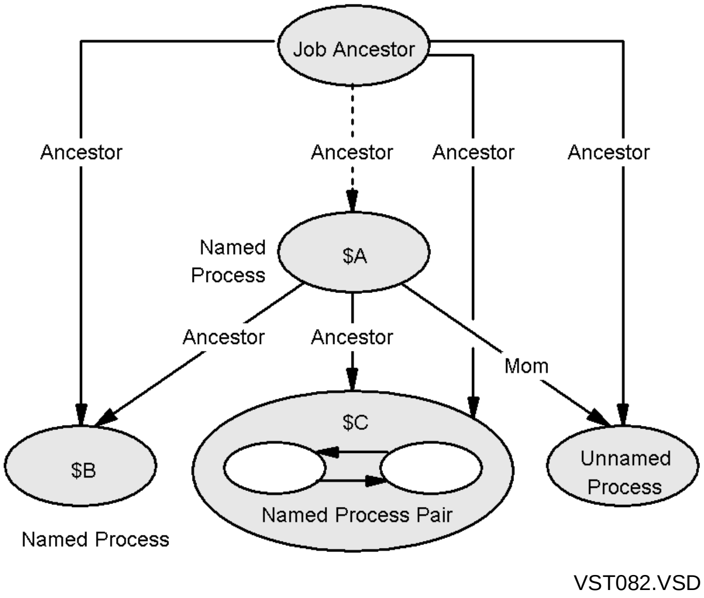
- The job ID for this job
- The process handle of the job ancestor

When a process that is part of a job gets deleted, the operating system sends a process-deletion message to the job ancestor as well as to the deleted process's creator. In this way, the job ancestor can keep track of which processes are still running.

Once again, because the linkage to the job ancestor is kept in the PCB of the process being deleted, the job ancestor does not receive the Process deletion message if the process is deleted because of a CPU failure. This is true even for named processes. The job ancestor must therefore monitor all CPUs where it has processes running and check for any CPU down messages.

Figure 16-3 shows job ancestor relationships.

Figure 16-3. Job Ancestor Relationships


Job Ancestor
Ancestor
Ancestor
Ancestor
Ancestor
Named Process
$A
Ancestor
Ancestor
Mom
$B
Named Process
$C
Named Process Pair
Unnamed Process
VST082.VSD

Relationship With a Home Terminal

Associated with each process is its home terminal. The IN and OUT files for the process is, by default, the home terminal.

The home terminal is usually the same as for the creator of the process; it is passed to a new process when the process is created. However, you can specify the home terminal in one of the following ways:

- Manually during process creation using the RUN command by specifying the terminal in the TERM parameter
- Programmatically during process creation using the PROCESS_LAUNCH_ or PROCESS_CREATE_ procedure
- Programmatically while the process is running using the PROCESS_SETSTRINGINFO_ procedure

For details on the RUN command, see the *TACL Reference Manual*. Details for setting the home terminal programmatically are given later in this section.

Process Subtype

The Guardian process subtype is an attribute that can be set at compile/bind time. A terminal-simulation program, for example, needs to be assigned process subtype 30 to allow it to assign itself a terminal-device type. See Section 24, Writing a Terminal Simulator, for an example.

All HP compilers that are not native and the HP linkers let you set the process subtype.

The default process subtype is zero. Other process subtypes are either available for your use or reserved as follows:

- Subdevice types 48 through 63 are available for your use.
- Subdevice types 1 through 47 are reserved. Extra protection is applied for subdevice types 1 through 15. To assign a subdevice type in this range, you must be the super ID user, be licensed, or have a PROGID that gives you super ID user status. Any other access yields an illegal process subtype error.

Process Priority

All processes on a CPU share the same priority structure, including system processes and application processes. It is therefore important that each process runs with a priority that permits necessary system operations when needed.

For example, if an application process initiates a nowait I/O operation against a disk file, it is important that the disk process runs with a higher priority than the application. Otherwise, the disk process would not take control of the CPU until the application process issues the AWAITIO procedure to wait for the completion of the I/O operation.

Table 16-1 provides an overview of suggested priority values for system and user processes. System process priorities are set during system generation as described in the *System Generation Manual* for D-series releases and in the *System Generation Manual for G-Series Releases* for G-series releases; I/O processes (but not disk processes) can also have their priorities dynamically changed using the Dynamic System Configuration (DSC) utility as described in the *Dynamic System Configuration (DSC) Manual* for D-series release systems; for G-series release systems, see the Subsystem Control Facility (SCF) and the SCF reference manuals. Application process priorities are set during process creation; see Creating Processes later in this section.

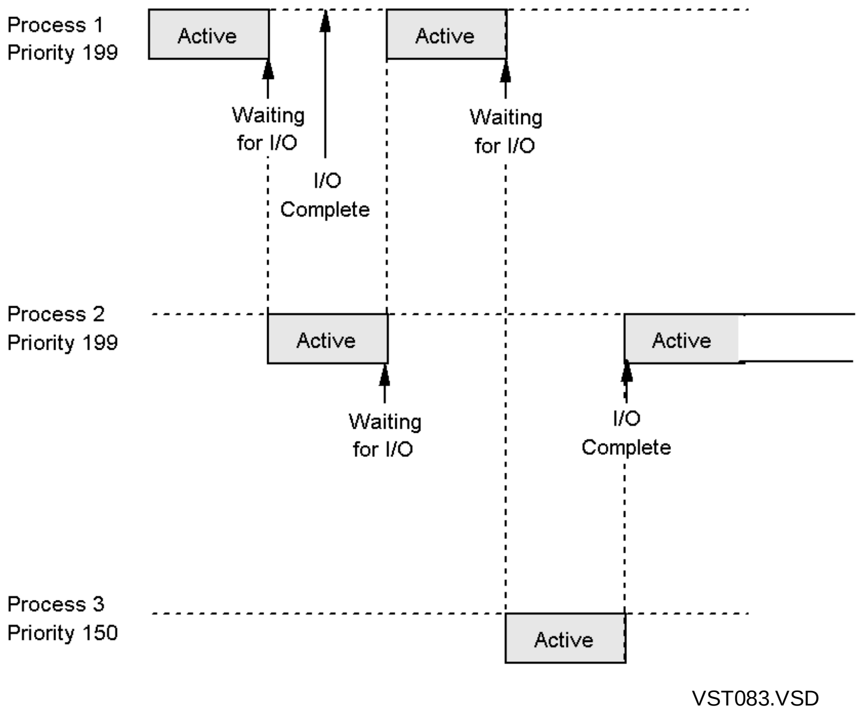
CPU-bound processes may have their priority reduced automatically to allow other processes to gain access to the IPU.

Table 16-1. Priority Values for System and User Processes

System Process	Priority	User Process	Priority
Disk I/O processes	220		.
TMF monitor	211		.
Memory manager	210		.
Operator process, $NCP, monitor, I/O processes other than disk, and so on	209 . . 200		. . . 200
$TMP	190	TACL processes used to run application processes	199 . .
		Application processes	150
		Application processes and editors used for program development (Priority 149 is assigned automatically by TACL processes running at priority 150)	149 . . 145
		Spoolers used for program development	144 . . 140
		Compilers and background batch processing	139 . .

The example in Figure 16-4 shows shows how CPU time is divided among three processes executing in the same IPU.

Figure 16-4. Execution Priority Example


Process 1
Priority 199
Active
Waiting for I/O
I/O Complete
Active
Waiting for I/O
Process 2
Priority 199
Active
Waiting for I/O
Active
I/O Complete
Process 3
Priority 150
Active
VST083.VSD

Notice that CPU time alternates between the two processes with priority 199. When one process is suspended for I/O, the other process runs.

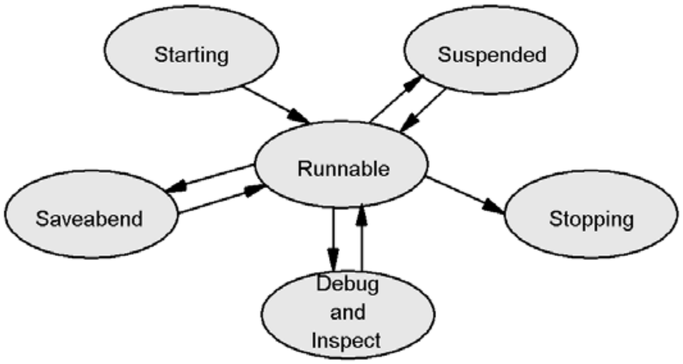
The only time that the process with priority 150 executes is when both of the other processes are suspended. Additionally, the lower-priority process is immediately suspended when a higher-priority process becomes ready.

This example does not account for the effects of system processes, nor does it show floating priorities.

Process Execution

From creation to deletion, a process enters and leaves several process states. Figure 16-5 shows the various process states that a process can be in.

Figure 16-5. Process States



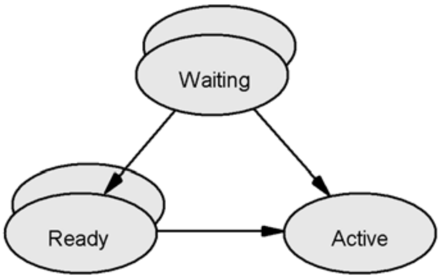
VST084.VSD

The states shown in Figure 16-5 are described as follows:

Debug and Inspect	Either the Debug or Inspect debugger has been invoked on the process. The Debug and Inspect state has several substates: memory access breakpoint (MAB), breakpoint (BKPT), trap (TRAP), and request (REQ). The trap state applies when debugging was entered because of a trap or signal. See Section 25, Debugging, Trap Handling, and Signal Handling, for information on process debugging.
Runnable	All process resources are available. The process is executing, ready to execute, or awaiting the services for the memory manager (because of a page fault).
Saveabend	The process is waiting for a SAVEABEND file to be created by the Inspect program.
Starting	This is a temporary state that occurs while the process is starting up.
Stopping	This is a temporary state that occurs while the process is being deleted.
Suspended	The process is waiting indefinitely as the result of a PROCESS_SUSPEND_ call. The process remains in this state until the PROCESS_ACTIVATE_ call makes the process runnable again.

The runnable state is of most interest. Only processes that are in the runnable state are capable of using the CPU. Figure 16-6 shows the possible substates of a runnable process.

Figure 16-6. Runnable Processes



VST085.VSD

The CPU is composed of one or more IPUs, each of which can execute a process. Associated with each IPU is a list of READY processes to be run on that IPU which is called a ready list. The process scheduler rearranges those lists occasionally for load-balancing and responsiveness purposes.

An **active process** is a process which is currently using an IPU. On a CPU with 4 IPUs there are four active processes. The process chosen to be the active process is the one with the highest priority on the given IPU's ready list.

The active process goes into the **waiting** state when it can no longer use the CPU, for example when waiting for an external event to complete such as an I/O operation. An active process goes into the **ready** state if it is preempted by a higher priority ready process.

Waiting processes that have satisfied their wait conditions go into the **ready** state.

The highest priority ready process in an IPU goes into the **active** state when

1) The active process on the IPU goes into the waiting state, or

2) The active process is of lower priority than the highest priority process in the given IPU's ready list.

Creating Processes

To programmatically create a Guardian process, you call the PROCESS_LAUNCH_ or PROCESS_CREATE_ procedure and pass it the name of the program file containing the program you want to execute. You can optionally supply a user library file providing additional procedures. The PROCESS_LAUNCH_ and PROCESS_CREATE_ procedures return a process handle that you can use to identify the created process in subsequent procedure calls.

The PROCESS_CREATE_ procedure does not allow you to specify values for some of the attributes that are associated with TNS/R native processes. Because it has less function than the PROCESS_LAUNCH_ procedure, PROCESS_CREATE_ is not discussed further in this section. (However, PROCESS_CREATE_ continues to be used in code examples in other sections of this guide.)

The PROCESS_SPAWN_ procedure allows you to create an Open System Services (OSS) process on the local or a remote TNS/R CPU, but it does not create Guardian processes.

To create processes interactively, you can use the RUN command. See the *TACL Reference Manual* for details.

The process creation examples in this section use the PROCESS_LAUNCH_ procedure. The PROCESS_LAUNCH_ procedures takes several parameters. The only required parameter is a structure that allows you to specify values establishing the following properties of the process:

- Whether the process will be named or unnamed
- Whether the creation will be a waited or nowait operation
- Other process attributes, including:
    - Whether the process will run at a high PIN or low PIN
    - The home terminal of the process
    - The size of the user data space (TNS processes only)
    - The size of the process file segment (PFS)
    - Any library file to be included in the process
    - The amount of swap space to be guaranteed the process by the Kernel-Managed Swap Facility (KMSF)
    - The device subtype of the process
    - The CPU where the process is to run
    - The way DEFINEs are propagated to the new process
    - The process priority

The following paragraphs describe how to create a process and, in doing so, how to set the above attributes.

Using the PROCESS_LAUNCH_ Procedure

The only required parameter to the PROCESS_LAUNCH_ procedure is a structure that allows you to specify values establishing the attributes of the new process. In the DLAUNCH file, which is located on $SYSTEM.SYSTEM, this structure is defined as follows:

```
STRUCT PROCESS_LAUNCH_PARMS_(*) FIELDALIGN(SHARED2);
BEGIN
INT         VERSION;      ! version of the structure
INT         LENGTH;       ! length of the structure
STRING .EXT PROGRAM_NAME;
INT(32)     PROGRAM_NAME_LEN;
STRING .EXT LIBRARY_NAME;
INT(32)     LIBRARY_NAME_LEN;
STRING .EXT SWAPFILE_NAME;
INT(32)     SWAPFILE_NAME_LEN;
STRING .EXT EXTSWAPFILE_NAME;
INT(32)     EXTSWAPFILE_NAME_LEN;
STRING .EXT PROCESS_NAME;
INT(32)     PROCESS_NAME_LEN;
STRING .EXT HOMETERM_NAME;
INT(32)     HOMETERM_NAME_LEN;
STRING .EXT DEFINES;
INT(32)     DEFINES_LEN;
INT(32)     NOWAIT_TAG;
INT(32)     PFS_SIZE;
INT(32)     MAINSTACK_MAX;
INT(32)     HEAP_MAX;
INT(32)     SPACE_GUARANTEE;
INT(32)     CREATE_OPTIONS;
INT         NAME_OPTIONS;
INT         DEBUG_OPTIONS;
INT         PRIORITY;
INT         CPU;
INT         MEMORY_PAGES;
INT         JOBID;
INT         END_NOV95[0:-1];  ! dummy, to determine size of
                              ! current version of the struct
END;
```

The DLAUNCH file also defines literals for the VERSION and LENGTH fields (the first two fields of the structure) and includes an array of default values (P_L_DEFAULT_PARMS) for initalizing the input parameter structure. The DLAUNCH file is for use with TAL and pTAL programs. The DLAUNCHH file is provided for use with C and C++ programs.

If the value of a pointer field is equal to NIL_ or if the corresponding length parameter is equal to 0, the item is considered to be omitted. The literal NIL_ is defined in the DLAUNCH and DLAUNCHH files.

Declarations supporting the PROCESS_LAUNCH_ procedure, including a definition of the input parameter structure, are also contained in the ZSYSTAL file. For C and C++

programs, declarations are contained in the ZSYSC file. For further information about using the ZSYS* files, refer to the subsection Using Parameter Declarations Files in Section 1, Introduction to Guardian Programming.

The main output of the PROCESS_LAUNCH_ procedure is returned by a parameter that is also formatted as a structure. Its format is identical to that of the nowait PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message (system message -102). In the TAL ZSYSTAL file, the structure for this message is defined as follows:

```
STRUCT       ZSYS^DDL^SMSG^PROCCREATE^DEF (*)
   BEGIN
   INT          Z^MSGNUMBER;
   INT(32)      Z^TAG;
   STRUCT       Z^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   INT          Z^ERROR;
   INT          Z^ERROR^DETAIL;
   INT          Z^PROCNAME^LEN;
   INT          Z^RESERVED[0:3];
   STRUCT       Z^DATA;
      BEGIN
      FILLER       50;
      END;
   STRUCT       Z^PROCNAME = Z^DATA;
      BEGIN STRING BYTE [0:49]; END;
   END;
```

Creating an Unnamed Process

Remember that processes can be named or unnamed. To create an unnamed process, you call the PROCESS_LAUNCH_ procedure with the NAME_OPTIONS field set to 0 in the input parameter structure. To set the NAME_OPTIONS field, you can use the ZSYS^VAL^PCREATOPT^NONAME literal from the ZSYSTAL file. All you need to supply is the program file name.

The following example creates an unnamed process:

```
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(PROCESS_LAUNCH_);
?SOURCE $SYSTEM.SYSTEM.DLAUNCH(PROCESS_LAUNCH_DECS);
?SOURCE $SYSTEM.SYSTEM.ZSYSTAL;
?LIST
 .
 .

STRING PROG_NAME[0:ZSYS^VAL^LEN^FILENAME-1];
INT .EXT ERROR_DETAIL,
         OUTPUT_LIST_LEN;
STRUCT OUTPUT_LIST(ZSYS^DDL^SMSG^PROCCREATE^DEF);
STRUCT PARAM_LIST(PROCESS_LAUNCH_PARMS_);
 .
 .
PARAM_LIST ':=' P_L_DEFAULT_PARMS_; ! initialize param struct
PROG_NAME ':=' "PROGFILE" -> @S^PTR;  ! program file name
@PARAM_LIST.PROGRAM_NAME := $XADR(PROG_NAME);
PARAM_LIST.PROGRAM_NAME_LEN := $DBL(@S^PTR '-' @PROG_NAME);
PARAM_LIST.NAME_OPTIONS := ZSYS^VAL^PCREATOPT^NONAME;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

In this example, the OUTPUT_LIST structure returns the process handle and the unnamed form of the process descriptor, which is suitable for supplying as the `file-name` parameter to the FILE_OPEN_ procedure. See Section 2, Using the File System, for a discussion of process file names.

Creating a Named Process

When creating a named process, you can either specify the process name yourself or have the operating system assign a name for you.

Specifying a Process Name

The next example is in C. To specify a process name, you must set the name_options field to 1 and supply the process name in the process_name field in the input parameter structure. To set the name_options field, you can use the ZSYS^VAL^PCREATOPT^NAMEINCALL literal.

The following example supplies a process name:

```
#include <dlaunch.h>
#include "$system.zsysdefs.zsysc(zsys_ddl_smsg_proccreate, \
                                 process_constant)"
#include <cextdecs(PROCESS_LAUNCH_)>
 .
 .
process_launch_parms_def paramList = P_L_DEFAULT_PARMS_;
zsys_ddl_smsg_proccreate_def outputList;
short error, errorDetail, outputListLen;
 .
 .
paramList.program_name = "PROGFILE";
paramList.program_name_len = sizeof("PROGFILE") - 1;
paramList.name_options = ZSYS_VAL_PCREATOPT_NAMEINCALL;
paramList.process_name = "$REQ";
paramList.process_name_len = sizeof("$REQ") - 1;
error = PROCESS_LAUNCH_( &paramList,
                         &errorDetail,
                         &outputList, sizeof(outputList),
                         &outputListLen);
```

Here, the information returned in the outputList structure includes the named form of a process descriptor. See Section 2, Using the File System, for a discussion of process names and process descriptors.

Requesting a System-Generated Process Name

Use either of the following operations to make the system provide a name for your process:

- Use the PROCESSNAME_CREATE_ procedure to create the name. You can pass the name to the PROCESS_LAUNCH_ procedure the same way you would a user-specified name.
- Call the PROCESS_LAUNCH_ procedure with the NAME_OPTIONS field of the input parameter structure set to the value of the ZSYS^VAL^PCREATOPT^NAMEDBYSYS literal.

If you need the system to create a process name before you create the corresponding process, or if you need the system to create a remote long name (5 characters plus the $ sign), then you must use the PROCESSNAME_CREATE_ procedure. Otherwise, you can use PROCESS_LAUNCH_ with the NAME_OPTIONS field of the input parameter structure set for a system-generated name. Each of these methods is described in the following paragraphs.

Using the PROCESSNAME_CREATE_ Procedure

The PROCESSNAME_CREATE_ procedure gives you the option of returning a 6-character process name or a 5-character process name, and you have the option of adding a node name. You use the *`name-type`* parameter to request the length of the name. You use the *`options`* parameter and, if appropriate, the *`nodename`*`:`*`length`* parameter to specify whether you want to add a node name to the process name.

The following example requests a 6-character process name to include a node name. The name is returned in the PROCESS^NAME parameter. This, and the remaining examples are in TAL.

```
LITERAL SIX^CHARACTERS   = 1,
        INCLUDE^NODENAME = 0;
  .
  .
MAX^LENGTH := ZSYS^VAL^LEN^PROCESSDECR;
NAME^TYPE := SIX^CHARACTERS;
NODENAME ':=' "\CENTRAL" -> @S^PTR;
NODENAME^LENGTH := @S^PTR '-' @NODENAME;
OPTIONS := INCLUDE^NODENAME;
CALL PROCESSNAME_CREATE_(PROCESS^NAME:MAX^LENGTH,
                         NAME^LENGTH,
                         NAME^TYPE,
                         NODENAME:NODENAME^LENGTH,
                         OPTIONS);
```

You can pass the name returned in PROCESS^NAME to the PROCESS_LAUNCH_ procedure as a user-supplied name.

Using the PROCESS_LAUNCH_ Procedure

To have the system supply a name without using the PROCESSNAME_CREATE_ procedure, you can call the PROCESS_LAUNCH_ procedure with the NAME_OPTIONS field set equal to the ZSYS^VAL^PCREATOPT^NAMEDBYSYS literal in the input parameter structure. The information returned in the output parameter structure includes a process descriptor, suitable for passing to the FILE_OPEN_ procedure, and the length of the descriptor.

The operating system supplies a name for a new process in the following example:

```
  .
  .
PROG_NAME ':=' "REQFILE" -> @S^PTR;
@PARAM_LIST.PROGRAM_NAME := $XADR(PROG_NAME);
PARAM_LIST.PROGRAM_NAME_LEN := $DBL(@S^PTR '-' @PROG_NAME);
PARAM_LIST.NAME_OPTIONS := ZSYS^VAL^PCREATOPT^NAMEDBYSYS;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Note. If the “run named" object-file flag is set (at compile or link time), then the system generates a name even if the NAME_OPTIONS field in the input parameter structure is set to 0.

Creating a Process in a Nowait Manner

If you call the PROCESS_LAUNCH_ procedure with the NOWAIT_TAG field of the input parameter structure set to any value other than -1, your process returns immediately without waiting for completion of the operation. Instead, your process receives notification with system message -102 (the nowait PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message) when the operation finishes.

The format of system message -102 was shown earlier as it is defined in the ZSYSTAL file; its format is the same as that of the PROCESS_LAUNCH_ output parameter structure. The structure of system message -102 is shown below as an array:

Structure of the PROCESS_LAUNCH or PROCESS_CREATE_ completion message (-102):

```
sysmsg[0]                  = -102
sysmsg[1] FOR 2            = Nowait tag supplied to
                             process creation procedure
sysmsg[3] FOR 10           = Process handle of the new process
sysmsg[13]                 = Error
sysmsg[14]                 = Error detail
sysmsg[15]                 = Length of process descriptor for
                             new process
sysmsg[16] FOR 4           = Reserved for future use
sysmsg[20] FOR sysmsg[15] = Process descriptor of new process
```

The following example creates a named process in a nowait manner. By setting the NOWAIT_TAG field to 1 in the input parameter structure, the PROCESS_LAUNCH_ procedure returns immediately without returning any value for the process handle or the process descriptor in the output parameter structure. Instead, these values are retrieved from system message -102 by calling the READUPDATEX procedure on the $RECEIVE file:

```
  .
  .
PROG_NAME ':=' "REQFILE" -> @S^PTR;
@PARAM_LIST.PROGRAM_NAME := $XADR(PROG_NAME);
PARAM_LIST.PROGRAM_NAME_LEN := $DBL(@S^PTR '-' @PROG_NAME);

PARAM_LIST.NAME_OPTIONS := ZSYS^VAL^PCREATOPT^NAMEINCALL;
PROC_NAME ':=' "$REQ" -> @S^PTR;
@PARAM_LIST.PROCESS_NAME := $XADR(PROC_NAME);
PARAM_LIST.PROCESS_NAME_LEN := $DBL(@S^PTR '-' @PROC_NAME);

PARAM_LIST.NOWAIT_TAG ':=' 1D;
```

```
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL);

IF ERROR <> 0 THEN ...
 .
 .

CALL READUPDATEX(RCV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF <> THEN ...

IF BUFFER[0] = -102 THEN                       !Process create
BEGIN                                          ! completion message
   IF BUFFER [13] <> 0 THEN ...                !Error
   ELSE
   BEGIN
      NOWAIT^TAG := BUFFER[1] FOR 2;
      PROCESS^HANDLE ':='
         BUFFER[3] FOR ZSYS^VAL^PHANDLE^WLEN;
      PROCESS^DESCRIPTOR^LENGTH := BUFFER[15];
      PROCESS^DESCRIPTOR ':=' BUFFER[20] FOR
                              PROCESS^DESCRIPTOR^LENGTH;
   END;
END;
```

The returned nowait tag enables you to match the message with the corresponding call to PROCESS_LAUNCH_.

Analyzing Process-Creation Errors

If your process creation fails, you will receive an error indication in the returned *`error`* value. An additional level of detail is returned in the *`error_detail`* parameter. For a waited creation attempt, these variables are returned by the PROCESS_LAUNCH_ call. For a nowait creation attempt, the error variables are returned in system message -102.

See the *Guardian Procedure Errors and Messages Manual* for a list of each possible value of *`error`* and an interpretation of the associated *`error_detail`* value.

Waited Creation Errors

If you call PROCESS_LAUNCH_ in a waited manner, you can gather any error information as soon as the call returns. In addition to the *`error`* value returned, you also get an *`error_detail`* parameter. The information returned in *`error_detail`* depends on the value in *`error`*. For example, if *`error`* is 1, then PROCESS_LAUNCH_ encountered a file-system error; *`error_detail`* indicates which file-system error.

The following example examines the *`error_detail`* parameter:

```
 .
 .
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
IF ERROR <> 0 THEN
BEGIN
   CASE ERROR OF
   BEGIN
   '1' -> CALL FILE^ERRORS(ERROR_DETAIL); !To process the
                                          ! file-system error

   '2' -> CALL PARAM^ERROR(ERROR_DETAIL); !To process the
                                          ! parameter error
    .
    .
   END;
END;
```

Nowait Creation Errors

If you call PROCESS_LAUNCH_ in a nowait manner, you need to check not only the error return value of the procedure call but also the PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message. If the system is unable to initiate process creation (for example, if you specified an invalid IPU number), then the system returns the error with the procedure call. Other process-creation errors are reported in the PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message.

Specifying Process Attributes and Resources

The following paragraphs show how to set process attributes and resources when you create a new process with the PROCESS_LAUNCH_ procedure.

Running a Process at a High PIN or a Low PIN

You can run a process at a high PIN (256 or greater) or low PIN (254 or less). Whether to run processes at a high PIN or low PIN depends on how many PINs your system will need. Although some HP processes will use high PINs, many will use low PINs. Consequently, you should consider running application processes at high PINs, because there are many more high PINs available than there are low PINs. The danger of running a process at a low PIN is that if the system uses all the available low PINs, you will not be able to run all the processes you want.

Whether a new process runs at a high PIN or a low PIN depends on:

- The force-low flag—bit 31 in the CREATE_OPTIONS field of the input parameter structure of the PROCESS_LAUNCH_ procedure call
- The inherited force-low characteristic of the creator process
- The ignore force-low flag—bit 26 in the CREATE_OPTIONS field of the input parameter structure of the PROCESS_LAUNCH_ procedure call
- The HIGHPIN attribute of the program file and the library file or SRL file, if there is one

Figure 16-7 summarizes how these conditions affect whether new processes run at a high PIN. A description of each of these entities follows.

The Force-Low Flag

If you set bit 31 (the force-low flag) to 1 in the CREATE_OPTIONS field of the input parameter structure when you call PROCESS_LAUNCH_, the new process will run at a low PIN. (Note that processes started with the C-series NEWPROCESS procedure always run at a low PIN.)

The Inherited Force-Low Characteristic

If the inherited force-low characteristic of your process is set, the new process normally runs at a low PIN. This flag is contained within the PCB and is normally inherited from the creator. A process has its inherited force-low characteristic set if one of the following is true:

- The creator was created with bit 31 (the force-low flag) set to 1 in the CREATE_OPTIONS field.
- The creator was created using the C-series NEWPROCESS procedure.
- The creator inherited the force-low characteristic from its creator.

The Ignore Force-Low Flag

To override the inherited force-low characteristic, you set bit 26 (the ignore force-low flag) to 1 in the CREATE_OPTIONS field of the input parameter structure. As a result, the new process can run at either a high PIN or a low PIN, depending upon the force-low flag and the program, library, and SRL files. In addition, the new process does not have its inherited force-low characteristic set.

The HIGHPIN Attribute

If the HIGHPIN attribute is set for the program file and for any library file or SRL file, the new process can run at a high PIN. To do so, however, the process must not be forced into a low PIN by either CREATE_OPTIONS.<31>, or the inherited force-low characteristic.

The following example shows one way of setting the HIGHPIN file attribute, using the BINDER program:

```
28> BIND CHANGE HIGHPIN ON IN tnsobj
```

The next example shows the use of the `nld` utility to perform the same action on an existing TNS/R native object file:

```
29> nld -change highpin on natobj
```

Both utilities can also set attributes while creating the object file. The following example shows how to set the HIGHPIN attribute while linking two native object files:

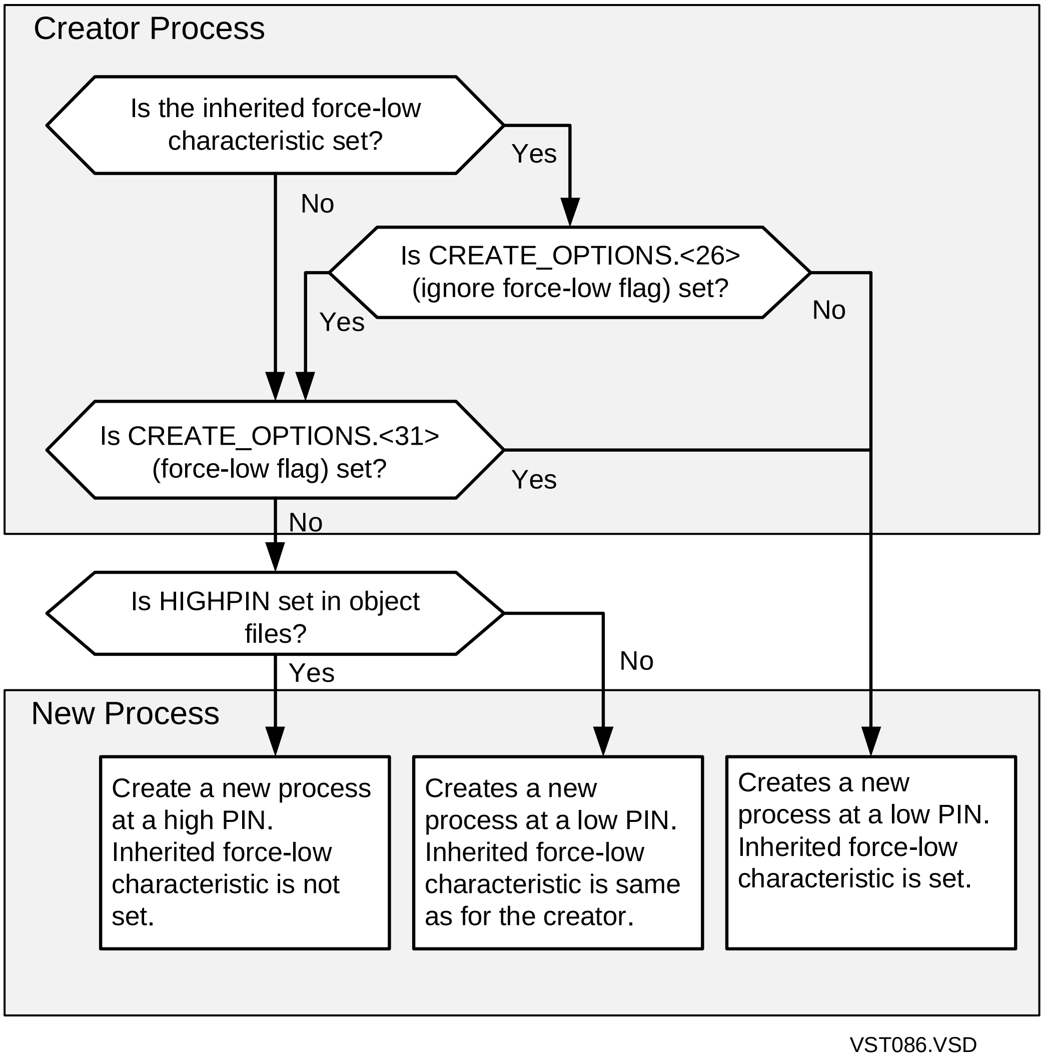
```
30> nld ofile1 ofile2 -set highpin on -o objfile
```

The `nld` and Binder utilities only need to set this attribute once, either after building the object file (using the CHANGE command) or while building the object file (using the SET command).

△ **Caution.** Some C-series system procedures (such as MYPID) do not support high PINs. If your program contains any such calls then the process will stop with a run-time error. By using the D-series procedure calls described in this manual, and in the *Guardian Procedure Calls Reference Manual*, your process will be able to run at a high PIN.

If your program does contain superseded C-series system procedure calls, see the *Guardian Application Conversion Guide* for a description of how to enable the process to run at a high PIN.

Figure 16-7. Running a Process at a High PIN or a Low PIN


Creator Process
Is the inherited force-low characteristic set?
Yes
No
Is CREATE_OPTIONS.<26> (ignore force-low flag) set?
Yes
No
Is CREATE_OPTIONS.<31> (force-low flag) set?
Yes
No
Is HIGHPIN set in object files?
Yes
No
New Process
Create a new process at a high PIN. Inherited force-low characteristic is not set.
Creates a new process at a low PIN. Inherited force-low characteristic is same as for the creator.
Creates a new process at a low PIN. Inherited force-low characteristic is set.
VST086.VSD

Starting a High-PIN Process Programmatically: An Example

The following code fragment creates a new process to run at a high PIN or low PIN depending on the inherited force-low characteristic. Note that CREATE_OPTIONS.<31> is set to 0 in the input parameter structure. This example assumes that the HIGHPIN file attribute is set in the program file of the process that you are creating:

```
  .
  .
PARAM_LIST.CREATE_OPTIONS:= 0D; !bits 26 and 31 both set to 0
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Starting a High-PIN Process Interactively: Examples

The next example causes the TACL program to start any new processes at a high PIN:

```
3> SET HIGHPIN ON
```

The SET HIGHPIN ON command causes the TACL process to create any new process with the force-low flag set to 0, allowing processes to be created at a high PIN, if the HIGHPIN attribute is set in the object file being executed. (A TACL process always runs with the ignore force-low flag set to 1.)

The following example uses the TACL run option to cause the program to run at a high PIN, if possible.

```
20> RUN objfile /HIGHPIN ON/
```

Specifying the Home Terminal

By default, your process receives input from its home terminal and sends output to its home terminal; that is, the home terminal name serves as the default value for the IN and OUT parameters of the process Startup message (see Section 8, Communicating With a TACL Process). Normally, the home terminal for a new process is the same as for the creator process. However, you can choose a different home terminal for your new process by supplying values for the HOMETERM_NAME and HOMETERM_NAME_LEN fields of the input parameter structure when calling the PROCESS_LAUNCH_ procedure.

The following example specifies the home terminal:

```
  .
  .
TERM_NAME ':=' "$TERM1" -> @S^PTR;
@PARAM_LIST.HOMETERM_NAME = $XADR(TERM_NAME);
PARAM_LIST.HOMETERM_NAME_LEN := $DBL(@S^PTR '-' @TERM_NAME);
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                                 OUTPUT_LIST:$LEN(OUTPUT_LIST),
                                 OUTPUT_LIST_LEN);
```

Sizing the TNS User Data Area

If you are creating a TNS process, you can use any of the following methods to specify the number of pages of user data area that your new process can occupy:

- The DATAPAGES compiler directive or pragma; see the appropriate compiler manual for details.
- The Binder program; see the *Binder Manual* for details.
- The RUN command MEM option; see the *TACL Reference Manual* for details.
- The MEMORY_PAGES field of the input parameter structure of the PROCESS_LAUNCH_ procedure.

The compiler or Binder value sets the minimum number of data pages. You can increase this number using the RUN command or the PROCESS_LAUNCH_ procedure, but you cannot reduce this number. The number that you supply is in disk pages, which are 2048 bytes each. The actual amount of memory allotted to the user data area is rounded up to the nearest multiple of the memory page size for your system, which is 4 KB (or two disk pages) for NSR-L CPUs, and 16 KB (or eight disk pages) for all other CPU types.

The following example uses the PROCESS_LAUNCH_ procedure to create a process with five user data pages:

```
  .
  .
PARAM_LIST.MEMORY_PAGES := 5;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

On an NSR-L CPU (such as a K1000 system), 6*2048 bytes (or three memory pages) will be allocated. For other CPUs, the memory page size is 16 KB, so 8*2048 bytes (or one memory page) will be allocated.

The maximum number of data pages allowed is 64.

Sizing the Process File Segment

Another memory dimension you can set using the PROCESS_LAUNCH_ procedure is the size of the PFS. You specify the PFS size in bytes using the PFS_SIZE field of the input parameter structure.

One of the functions of the PFS is to provide space for managing files and DEFINEs. The larger the PFS is, the more files the process can have open. The default PFS size of 256 KB is enough for most processes. However, if you get a file system error 31 or

34, or a DEFINE error 2052, then you need extra PFS space for your program to run. You can create a PFS of up to one megabyte.

If you need to reduce the amount of virtual memory used by your process, you might be able to reduce the PFS size to a minimum of 128 KB.

You can use a linker (either the Binder or the `nld` utility) to specify the size of the PFS. (See the *Binder Manual* or the *nld and noft Manual*). The value set by the linker becomes the default value for your process. To create a different-sized PFS in your program, you must specify the PFS size in bytes. This value must change the PFS size to something between one and eight unitary segments (131,072 to 1,048,576 bytes).

The following example sets the process file segment size to 524,288 bytes:

```
  .
  .
PARAM_LIST.PFS_SIZE := 524288D;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Specifying a User Library File

Use the LIBRARY_NAME and LIBRARY_NAME_LEN fields of the input parameter structure of the PROCESS_LAUNCH_ procedure to specify a procedure library. This library is a second object file containing user-written procedures called from the program file. When a TNS process is created, the library file occupies the user library space (UL). When a native process is created, the library file occupies the native UL space.

The following example uses a library file named PROCLIB:

```
  .
  .
LIB_NAME ':=' "PROCLIB" -> @S^PTR;
@PARAM_LIST.LIBRARY_NAME := $XADR(LIB_NAME);
PARAM_LIST.LIBRARY_NAME_LEN := $DBL(@S^PTR '-' @LIB_NAME);
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Specifying Swap Files

In a large majority of cases, it is unnecessary to specify a swap file for a new process, whether by supplying a value in the SWAPFILE_NAME field or in the EXTSWAPFILE_NAME field of the input parameter structure. In general, you should simply allow the system to manage swap space.

For all processes, values supplied in the SWAPFILE_NAME and SWAPFILE_NAME_LEN fields of the input parameter structure of the

PROCESS_LAUNCH_ procedure are unused except for information purposes (that is, to support programs that use the swap file name to determine the volume on which to create temporary files). The actual swap space is handled by the Kernel-Managed Swap Facility (KMSF), regardless of whether these fields are used.

If your TNS process uses a default extended data segment, you can use the EXTSWAPFILE_NAME and EXTSWAPFILE_NAME_LEN fields of the input parameter structure of the PROCESS_LAUNCH_ procedure to specify a swap file for that segment. (These fields are ignored for TNS/R native processes.) Specifying a swap file for a TNS process in this manner is supported for compatibility, but it is not recommended. For best performance, you should allow the system to use KMSF to manage swap space.

Requesting Guarantee of Swap Space From KMSF

Most swap space is handled by the Kernel-Managed Swap Facility (KMSF). For each CPU, KMSF manages one or more swap files from which swap space is allocated for the processes in that CPU.

A process is automatically allocated swap space by KMSF as needed. However, if you want to ensure that a particular amount of swap space is available for your process, you can specify a value (other than 0) in the SPACE_GUARANTEE field of the input parameter structure when calling PROCESS_LAUNCH_. (You can also set the space guarantee value using the `nld` utility; refer to the *nld and noft Manual*. However, you cannot set this attribute in a TNS object file.) KMSF reserves the amount of space specified in this field, in bytes, as swap space for the new process. The number of bytes is rounded up to the page size of the CPU.

The space guarantee applies to space allocated for the stack, the globals-heap segment, and any SRL data segments; it does not apply to space for extended data segments. (For information on allocating space for an extended data segment, refer to Allocating Extended Data Segments in Section 17, Managing Memory.) If KMSF cannot guarantee the amount of space requested, PROCESS_LAUNCH_ returns error 55.

Most processes do not need to set the space guarantee attribute, because KMSF allocates space as processes need it. Setting large guarantees on many processes could have a detremental effect on swap space consumption. The guarantee mechanism is provided for programs (such as some NonStop process pairs) that need to ensure, when starting, that they will not later fail due to competition for resources.

For more information about KMSF, refer to the *Kernel-Managed Swap Facility (KMSF) Manual.*

The following example specifies that 262,144 bytes of swap space (equivalent to two segments) be reserved.

```
  .
  .
PARAM_LIST.SPACE_GUARANTEE := 262144D;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                                 ERROR_DETAIL,
                                 OUTPUT_LIST:$LEN(OUTPUT_LIST),
                                 OUTPUT_LIST_LEN);
IF ERROR <> 0 THEN ...
```

Specifying a Device Subtype

You can assign a device subtype attribute to a process at compile or link time. (There is no input parameter to PROCESS_LAUNCH_ for specifying a subtype attribute.) One use for giving a process a device subtype is when creating a terminal simulation process as described in Section 24, Writing a Terminal Simulator.

The HP linkers (the `nld` and Binder utilities) and the TNS compilers provide directives that allow you to assign a device subtype to a program file. For details, see the appropriate compiler manual; also see the *nld and noft Manual* or the *Binder Manual*. For H-series software, see the *nld Manual* and the *noft Manual* as well as the *Binder Manual*

Each subtype is an integer value. There are 64 possible values:

- Subtype zero is the default value.
- Subtypes 1 through 47 are reserved for HP use.
- Subtypes 48 through 63 are available for general use; you can create a named process with a device subtype in this range.

Specifying a CPU

Normally, a new process runs on the same CPU as its creator process. However, you can use the CPU field of the input parameter structure of the PROCESS_LAUNCH_ procedure to specify a CPU. The following example runs a new process on CPU number 6:

```
  .
  .
PARAM_LIST.CPU ':=' 6;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Specifying a New Job

For applications that do batch processing, you can use the JOBID field of the input parameter structure of the PROCESS_LAUNCH_ procedure to specify a new job.

For batch processing, you can group related processes into jobs by assigning the same job number to each process. The job number is interpreted by the NetBatch utility to establish the members of a given job.

When you start a job, you assign the job number to the first process in the job when you create the first process. The process that calls the PROCESS_LAUNCH_ procedure with a value specified in the JOBID field is known as the job ancestor (also known as the godmother or GMOM). When a process that is part of a job starts or terminates, the job ancestor receives a system message -112 (Job process creation) or a system message -101 (Process deletion), respectively. The job ancestor can use these messages to manage the job.

To assign a job ID to a process, you simply supply an integer value (other than 0 or -1) in the JOBID field of the input parameter structure of the PROCESS_LAUNCH_ procedure. The following example assigns a job ID of 1:

```
  .
  .
PARAM_LIST.JOBID := 1;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

When a process within a job creates an additional process, the job ID is normally passed on automatically to the additional process. For this to happen, the JOBID field of the input parameter structure of PROCESS_LAUNCH_ must be set to -1. When the additional process terminates, the system sends a death notification message to the job ancestor as well as to the creator of the process.

A process that is part of a job can start a process that does not belong to the job by issuing a PROCESS_LAUNCH_ procedure call with the JOBID field set to 0. Termination of such a process does not result in a death notification message being sent to the job ancestor, because the process is not a part of the job.

Propagating DEFINEs

Use the CREATE_OPTIONS and the DEFINES (and DEFINES_LEN) fields of the input parameter structure to indicate which DEFINEs in the environment of the current process should be propagated to the new process.

DEFINEs in the environment of the current process can be classified as:

- DEFINEs in the context of the process
- DEFINEs saved in a buffer by the process issuing the DEFINESAVE procedure call

Either or both of these groups of DEFINEs can be propagated to the new process. By default, only DEFINEs in the context of the current process are propagated to the new process; DEFINE mode in the new process is turned on or off as in the context of the current process. To change either of these default settings, use bits 27, 28, 29, and 30 of the CREATE_OPTIONS field of the input parameter structure as described below.

Bits 27 and 28 of the CREATE_OPTIONS field specify which DEFINEs are propagated.

- By default (bits 27 and 28 set to 0), only DEFINEs in the context of the current process are propagated to the new process.
- Set bit 28 to 1 and bit 27 to 0 to propagate DEFINEs saved by the current process. The address of the propagated DEFINE is passed in the DEFINES field of the input parameter structure of the PROCESS_LAUNCH_ procedure call.
- Set bit 27 to 1 and bit 28 to 0 to propagate DEFINEs from the context of the calling process and the DEFINEs listed in the DEFINES field of the input parameter structure.

To override the default setting for the DEFINE mode of the new process, you need to set bit 29 to 1 in the CREATE_OPTIONS field of the input parameter structure. The DEFINE mode of the new process is then specified by bit 30: to enable DEFINEs in the new process, set bit 30 to 1; to disable DEFINEs, set bit 30 to 0.

The following example turns on DEFINE mode for the new process and propagates the DEFINEs saved in the DEFINE save buffer by a previous call to the DEFINESAVE procedure.

```
  .
  .
PARAM_LIST.CREATE_OPTIONS.<27:28> := 2;  !Propagate all
                                         ! DEFINEs
PARAM_LIST.CREATE_OPTIONS.<29> := 1;     !Use bit 30
PARAM_LIST.CREATE_OPTIONS.<30> := 1;     !Set DEFINE mode on

PARAM_LIST.DEFINE := <address of DEFINE save buffer>
          ! buffer contents supplied from call to DEFINESAVE
PARAM_LIST.DEFINE_LEN := <length of buffer>
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Note. When the primary process of a process pair creates its backup, all DEFINEs in the context of the primary process are propagated to the backup regardless of the settings of bits in the CREATE_OPTIONS field of the input parameter structure. If a value is specified in the DEFINE_NAME field, it is ignored.

When you create the new process, the DEFINE working set is initialized with the default attributes of CLASS MAP.

See Section 7, Using DEFINEs, for details on how to use DEFINEs.

Distinguishing the Recipient of the Process Deletion Message

Use the CREATE_OPTIONS field of the input parameter structure to specify whether the system should exhibit D-series behavior (the default action) or C-series behavior when delivering system message -101 (Process deletion) after the process you are creating terminates.

Set CREATE_OPTIONS.<25> to 0 to cause D-series behavior. With this option, the Process-deletion message is sent only to the caller of PROCESS_LAUNCH_ or to the instance of the process pair that contains the caller of PROCESS_LAUNCH_. An “instance” is any process in an unbroken chain of primary and backup processes. Every process that is part of an instance has the same sequence number.

Set CREATE_OPTIONS.<25> to 1 to cause C-series behavior. With this option, a process with the same name as the caller of PROCESS_LAUNCH_ receives the Process-deletion message.

The following example requests C-series behavior:

```
  .
  .
PARAM_LIST.CREATE_OPTIONS.<25> := 1;
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
```

Sending the Startup Sequence to a Process

Section 8, Communicating With a TACL Process, describes how the TACL process sends a startup sequence to a new process that it has just created. Processes started from a user-written process do not automatically receive a startup sequence. It is up to you to determine what the sequence will be and how to do it.

If your new process expects the standard startup protocol (for example, if the new process uses the INITIALIZER procedure), then you should issue a standard startup sequence, with messages in the same format as issued by the TACL process. This subsection describes how to do this.

When using the standard startup sequence, your program must perform the following sequence:

1. Create the Startup message using a data structure in the form of a Startup message. See Section 8, Communicating With a TACL Process, for details of the Startup message format.
2. Create any Assign or Param messages you will send to the new process.
3. Create the new process using, for example, the PROCESS_LAUNCH_ procedure.
4. Open the new process.
5. Send the Startup message to the new process using the WRITE[X] procedure on the open process file.
6. Optionally send Assign messages to the new process using the WRITE[X] procedure on the open process file.
7. Optionally send a Param message to the new process using the WRITE[X] procedure on the open process file.

8. Close the new process.

Sending and Receiving the Startup Message

The following example shows two processes. The first process creates the second process and then sends it the Startup message.

The first program is an extension of the program shown in Section 8, Communicating With a TACL Process, that receives the Startup message. This example first reads its own Startup message by calling INITIALIZER. The INITIALIZER procedure then calls the START^PROC procedure, which processes the Startup message and returns an array containing the open file number of the IN file, the open file number of the OUT file, the length of the Startup message, and the Startup message itself.

After returning from the INITIALIZER procedure, the first program creates the second process, opens the second process, and then sends a Startup message to the second process. In this case, the second process receives the same Startup message as the first process.

The code for the first process appears on the following pages.

```
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!Global parameters
LITERAL MAXLEN =
        ZSYS^VAL^LEN^FILENAME;       !Maximum file-name length
INT OUTNUM;                          !OUT file number
INT INNUM;                           !IN file number
INT .S^PTR;          !pointer to end of string (word address)

STRUCT .CI^STARTUP;                  !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULTS;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAM[0:529];
END;
INT MESSAGE^LEN;

? NOLIST
? SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,
?         WRITEX,PROCESS_LAUNCH_,PROCESS_STOP_,FILE_CLOSE_,
?         OLDFILENAME_TO_FILENAME_,FILE_GETINFO_)
? SOURCE $SYSTEM.SYSTEM.DLAUNCH(PROCESS_LAUNCH_DECS);
? LIST

!------------------------------------------------------------
! Procedure to save the Startup message in a global
! structure.
!------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,MATCH) VARIABLE;
INT .RUCB,.START^DATA,.MESSAGE,LENGTH,MATCH;

BEGIN

   CI^STARTUP.MSGCODE ':=' MESSAGE FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;

END;
```

```
!------------------------------------------------------------
! Procedure to perform initialization for the program.
!------------------------------------------------------------

PROC INIT;

BEGIN
   STRING .IN^NAME[0:MAXLEN - 1];  !string form of IN file
                                   ! name
   INT     INNAME^LEN;             !length of IN file
   STRING .OUT^NAME[0:MAXLEN - 1]; !string form of OUT file
                                   ! name
   INT     OUTNAME^LEN;            !length of OUT file
   INT     ERROR;

!  Call INITIALIZER to read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Convert 12-word file name from Startup message into a
!  variable-length string:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.INFILE.VOLUME,
                        IN^NAME:MAXLEN,
                        INNAME^LEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the input file.

   ERROR := FILE_OPEN_(IN^NAME:INNAME^LEN,INNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Convert the output file name:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.OUTFILE.VOLUME,
                        OUT^NAME:MAXLEN,
                        OUTNAME^LEN);
  IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the output file:

   ERROR := FILE_OPEN_(OUT^NAME:OUTNAME^LEN,OUTNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

END;
```

```
!------------------------------------------------------------
! Main procedure calls INIT to read and save the Startup
! message and open the IN and OUT files, creates a new
! process, then passes on its own Startup message to the
! new process.
!------------------------------------------------------------

PROC INITIAL MAIN;
BEGIN

   STRUCT PARAM_LIST(PROCESS_LAUNCH_PARMS_);
                  !PROCESS_LAUNCH_ input parameter struct
   STRUCT OUTPUT_LIST(ZSYS^DDL^SMSG^PROCCREATE^DEF);
                  !PROCESS_LAUNCH_ output parameter struct
   INT .EXT ERROR_DETAIL;
   INT .EXT OUTPUT_LIST_LEN; !length of PROCESS_LAUNCH output
                          ! parameter struct as returned
   INT      F^NUM;               !file number for process
                                 ! file
   INT      ERROR;
   STRING   PROGNAME[0:MAXLEN - 1]; !string form of program
                                 ! file name

!  Read and save the Startup message and open the IN and OUT
!  files:

   CALL INIT;

!  Start the new process:

   PARAM_LIST ':=' P_L_DEFAULT_PARMS_;  !initialize input
                                        ! parameter struct
   PROGNAME ':=' "$XCEED.DJCEGD10.ZNEW" -> @S^PTR;
   @PARAM_LIST.PROGRAM_NAME := $XADR(PROGNAME);
   PARAM_LIST.PROGRAM_NAME_LEN := $DBL(@S^PTR '-' @PROGNAME);
   PARAM_LIST.NAME_OPTIONS := ZSYS^VAL^PCREATOPT^NAMEDBYSYS;
   ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the new process:

   ERROR := FILE_OPEN_(
      OUTPUT_LIST.Z^PROCNAME:OUTPUT_LIST.Z^PROCNAME^LEN,
      F^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
```

```
!  Send the new process the Startup message.  The receiving
!  process reply with file-system error ZFIL^ERR^CONTINUE
!  indicating that it is ready to receive Assign messages.

   CALL WRITEX(F^NUM,CI^STARTUP,MESSAGE^LEN);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(F^NUM,ERROR);
      IF ERROR <> ZFIL^ERR^CONTINUE THEN CALL PROCESS_STOP_;
   END;

!  There are no Assign messages so close the new process:

   CALL FILE_CLOSE_(F^NUM);
END;
```

The new process receives the Startup message the same way as any process receives a Startup message from the TACL process. The process simply opens its $RECEIVE file and reads the message. The recommended way to do this is by calling the INITIALIZER procedure.

```
?INSPECT, SYMBOLS, NOCODE, NOMAP
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!Global parameters
LITERAL MAXLEN =
        ZSYS^VAL^LEN^FILENAME;      !Maximum file-name length
INT OUTNUM;                         !OUT file number
INT INNUM;                          !IN file number
STRING .S^PTR;                      !pointer to end of string

STRUCT .CI^STARTUP;                 !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULTS;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAM[0:529];
END;
INT MESSAGE^LEN;
```

```
? NOLIST
? SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,
?         PROCESS_STOP_,OLDFILENAME_TO_FILENAME_)
? LIST

!------------------------------------------------------------
! Procedure to save the Startup message in a global
! structure.
!------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,MATCH) VARIABLE;
INT .RUCB,.START^DATA,.MESSAGE,LENGTH,MATCH;

BEGIN

   CI^STARTUP.MSGCODE ':=' MESSAGE FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;

END;
```

```
!------------------------------------------------------------
! Procedure to perform initialization for the program.
!------------------------------------------------------------

PROC INIT;

BEGIN
   STRING .IN^NAME[0:MAXLEN - 1];  !string form of IN file
                                   ! name
   INT     INNAME^LEN;             !length of IN file
   STRING .OUT^NAME[0:MAXLEN - 1]; !string form of OUT file
                                   ! name
   INT     OUTNAME^LEN;            !length of OUT file
   INT     ERROR;

!  Call INITIALIZER to read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Convert 12-word file name from Startup message
!  into a variable-length string:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.INFILE.VOLUME,
                        IN^NAME:MAXLEN,
                        INNAME^LEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the input file:

   ERROR := FILE_OPEN_(IN^NAME:INNAME^LEN,INNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Convert the output file name:

   ERROR := OLDFILENAME_TO_FILENAME_(
                        CI^STARTUP.OUTFILE.VOLUME,
                        OUT^NAME:MAXLEN,
                        OUTNAME^LEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open the output file:

   ERROR := FILE_OPEN_(OUT^NAME:OUTNAME^LEN,OUTNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

END;
```

```
!-------------------------------------------------------------
! Main procedure calls INIT to read and save the Startup
! message and open the IN and OUT files, creates a new
! process, then passes on its own Startup message to the new
! process.
!-------------------------------------------------------------

PROC INITIAL MAIN;
BEGIN


!  Read and save the Startup message and open the IN and OUT
!  files:

   CALL INIT;
END;
```

Sending and Receiving Assign and Param Messages

You can send any number of Assign messages and a Param message to your new process. You can do this using the standard startup sequence protocol, thereby enabling your new process to read the Assign and Param messages with the INITIALIZER procedure. However, note again that you can send this information any way that you like, so long as the recipient process is able to interpret the format.

To use the standard startup protocol, your program must create messages in exactly the format of the Assign or Param messages and then send them to the new process using the WRITE[X] procedure. The recipient process reads these messages by calling INITIALIZER with the parameters set for reading Assign and Param messages. The action of the new process is identical to that taken by a process reading Assign or Param messages from the TACL process.

For details of the contents of the Assign and Param messages and for details of how to read Assign and Param messages, refer to Section 8, Communicating With a TACL Process.

Monitoring a Child Process

Once you have created a child process, you often need to make sure that the child process continues to run. You can check whether the child process has stopped by reading messages from the $RECEIVE file. The following messages indicate that a child process might have stopped:

-2 (Processor down message)
-100 (Remote processor down message)
-101 (Process deletion message)
-110 (Loss of communication with node)

A simple way to check whether the condition that caused one of these messages caused a specific process to stop is to use the CHILD_LOST_ procedure. You supply the CHILD_LOST_ procedure with the message read from $RECEIVE and the process handle of the child process that you wish to monitor. For example:

```
INT     BUFFER[0:511];
STRING .SBUFFER := @SBUFFER '<<' 1;
 .
 .
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
CALL FILE_GETINFO_(RECV^NUM, ERROR);
IF (ZFILE^ERR^SYSMESS = ERROR) THEN
  BEGIN
  ...
   IF (BUFFER = -2) OR (BUFFER = -100) OR (BUFFER = -101)
      OR (BUFFER = -110) THEN
   BEGIN
    ERROR := CHILD_LOST_(SBUFFER:BYTES^READ,PROCESS^HANDLE);
    IF ERROR = 4 THEN ...      !the specified process is lost
    IF ERROR = 0 THEN ...      !the specified process is still
                               ! running
   .
   .
 END;
END;
```

The returned value is 4 if the message identified the specified process as lost. This value is 0 if the message did not indicate that the specified process was lost.

Deleting Processes

Use the PROCESS_STOP_ procedure to delete processes. PROCESS_STOP_ allows you to delete your own process or delete another process so long as you have the authority to do so.

When you delete a process, the operating system sends a Process deletion message (message number -101) to the creator process indicating that the process no longer exists. If the process is part of a job, the operating system also sends the Process deletion message to the job ancestor.

For an unnamed process, the operating system sends the system message to the mom process as specified in the mom field in the PCB of the terminating process. For a named process, the operating system sends the system message to the ancestor process as indicated in the DCT.

The Process deletion message contains the following information:

Structure of the Process deletion message:

```
sysmsg[0]              -101
sysmsg[1] FOR 10       Process handle of terminated process
          WORDS
sysmsg[11] FOR 4       Process processor time in microseconds
           WORDS
sysmsg[15]             Process job ID, 0 if the process is
not
                       part of a job
sysmsg[16]             Completion code
sysmsg[17]             Termination information (0 if not
                       supplied)
sysmsg[18] FOR 6       SPI subsystem ID
           WORDS
sysmsg[24] FOR 10      Process handle of external process
           WORDS       causing termination (null if none)
sysmsg[34]             Length in bytes of termination text
                       (starting at sysmsg[41])
sysmsg[35]             Offset in bytes (from beginning of
                       message) of process file name of
                       terminated named process
sysmsg[36]             Length in bytes of process descriptor
of
                       terminated named process (or process
                       pair)
sysmsg[37].<0:14>      Reserved
sysmsg[37].<15>        Abend: death caused by abnormal
deletion
                       if 1, otherwise by normal deletion
sysmsg[38] FOR 3       Reserved
           WORDS
sysmsg[41] FOR         Termination text (up to 80 bytes),
zero
 sysmsg[34] WORDS      length if no termination text supplied

sysmsg[ ] FOR          Process file name of terminated named
 sysmsg[36] WORDS      process, zero length for an unnamed
                       process
```

Deleting Your Own Process

You can delete your own process by calling the PROCESS_STOP_ procedure without any parameters. The following statement stops the current process:

```
CALL PROCESS_STOP_;
```

In addition to stopping your process, you can set parameters in the PROCESS_STOP_ procedure to return additional information in the Process deletion message. This information includes whether the process was stopped normally or abnormally and completion code information. Abnormal termination and completion codes are described in the following paragraphs.

Abnormal Deletion

You can indicate abnormal deletion by setting the *`options`* parameter to 1 for abnormal deletion. The operating system sends out the Process deletion message with bit 15 of word 37 set to 1.

The following example deletes your process abnormally:

```
OPTIONS := 1;
CALL PROCESS_STOP_(!process^handle!,
                   !specifier!,
                   OPTIONS);
```

If the *`options`* parameter is zero or omitted, then the process terminates normally. With normal termination, bit 15 of word 37 of the Process deletion message defaults to zero.

Setting Completion Codes

When deleting your own process, you can return additional information in the Process deletion message by setting completion codes.

The *`compl-code`* parameter takes an integer value that is reported in word 16 of the Process deletion message for the recipient of the message to interpret. The number assigned to this parameter overrides the default values of 0 for normal deletion and various nonzero values for abnormal deletion. For more information on using completion codes, see Appendix C in the *Guardian Procedure Calls Reference Manual*.

If your process defines Subsystem Programmatic Interface (SPI) error numbers, your process can use the *`termination-info`*, *`spi-ssid`*, and *`text`*`:`*`length`* parameters of the PROCESS_STOP_ procedure to return detailed completion code information in the system message:

- The *`termination-info`* parameter contains an integer representing the SPI error number. This value is passed in word 17 of the system message.
- The *`spi-ssid`* parameter identifies the SPI subsystem. This information is passed in words 18 through 23 of the system message.
- The *`text`*`:`*`length`* parameter contains up to 80 bytes of text to be read by the message recipient. This information is passed in the system message beginning at word 41. The length (in bytes) of the text message is specified in word 34.

The following statement sends completion code information in the Process deletion message:

```
CALL PROCESS_STOP_(!process^handle!,
                   !specifier!,
                   !options!,
                       COMPLETION^CODE,
                       TERMINATION^INFO,
                       SPI^SUBSYSTEM^ID,
                       MESSAGE^TEXT:TEXT^LENGTH)
```

For details about SPI error numbers and SPI subsystem identifiers, refer to the *SPI Programming Manual*.

Programs not using SPI can use the *`termination-info`* and *`text`*`:`*`length`* parameters to report arbitrary integer and string values, respectively. TACL displays nonzero termination-info and non-empty text for processes that it runs.

Deleting Other Processes

You can delete another Guardian process by supplying the process handle to the PROCESS_STOP_ procedure. You are allowed to stop a user process if it has not made itself unstoppable and if one of the following conditions is true:

- The process has called SETSTOP to set the stop mode to 0, making it stoppable by anyone.
- A Safeguard access control list (ACL) associated with the process gives you permission to stop the process.
- Your process is locally authenticated or the process you are stopping is remotely authenticated, and one of the following conditions is true:
    - You have the same process access ID or creator access ID as the process.
    - You are the group manager for the process access ID or creator access ID of the process.
    - You are the super ID (255,255).

(Being locally authenticated on a system means either that the process has logged on by successfully calling USER_AUTHENTICATE_ (or VERIFYUSER) on the system or that the process was created by a process that had done so. A process is also considered local if it is run from a program file that has the PROGID attribute set.)

When you delete another process, the operating system sets the completion code in the Process deletion message to 6, indicating that the process was deleted by another process.

The following example shows a process deleting a process that it previously created (and which therefore has the same process access ID):

```
  .
  .
ERROR := PROCESS_LAUNCH_( PARAM_LIST,
                          ERROR_DETAIL,
                          OUTPUT_LIST:$LEN(OUTPUT_LIST),
                          OUTPUT_LIST_LEN);
IF ERROR <> 0 THEN ...
```

```
  .
  .
!Delete the process created earlier:
ERROR := PROCESS_STOP_(OUTPUT_LIST.Z^PHANDLE);
IF ERROR <> 0 THEN ...
```

In this case, the process issuing the PROCESS_STOP_ procedure call also receives the Process-deletion message because it is the creator of the process.

Using Stop Mode to Control Process Deletion

You can use the SETSTOP procedure to determine who has the authority to delete your process. This procedure sets the stop mode for the process as follows:

0 Any other process can stop your process.

1 Only processes qualified as previously described can stop your process. This is the default value.

2 No other process can stop your process. Only a privileged caller can set this mode.

If the attempt to stop the process is rejected because of the stop mode, an error is returned to the calling process and the stop request is queued until stop mode is reduced to the level at which the stop request is accepted:

- If a stop request passes the security checks but the target process is at stop mode 2, then the request is queued until the stop mode is reduced to 1 or 0. File-system error 638 is returned to the calling process.
- If a stop request fails the security checks for a process running at either stop mode 1 or stop mode 2, then the request is queued until the stop mode is reduced to 0. File-system error 639 is returned to the calling process.

The following example uses the SETSTOP procedure to set the stop mode of the calling process to 0;

```
LITERAL ANYONE^CAN^STOP^ME = 0;
  .
  .
CALL SETSTOP(ANYONE^CAN^STOP^ME);
```

A process can always stop itself, even if the stop mode is 2.

△ **Caution.** Any process using stop mode 2 when a trap or nondeferrable signal occurs will cause a IPU halt. Such a halt occurs, for example, if an unmirrored disk that contains a swap volume fails.

Reusing Resources Held by a Stopped Process

You need to be sure that a process has terminated before reusing any files the process had exclusive access to. If the PROCESS_STOP_ procedure returned error 0, 638, or 639, then the process might not yet have terminated and will not have released the files and devices it has exclusive access to. However, if the error returned is 0, then the process will not execute any more code.

The best way to ensure that the process is terminated and its resources freed is to wait for system message -101 (Process deletion), which is sent to its creator when the process terminates.

Suspending and Activating Processes

Remember that a process can alternate between the suspended and runnable states. You can cause a process to change from one state to the other either by issuing commands at the TACL prompt or programmatically by issuing system procedure calls.

You can suspend a runnable process by issuing the SUSPEND command at the TACL prompt; you can suspend a process programmatically by calling the PROCESS_SUSPEND_ procedure. To activate a suspended process, you can issue the TACL RESUME command, or you can activate a process programmatically by calling the PROCESS_ACTIVATE_ procedure.

This subsection describes the system procedure calls that suspend and activate processes. For a description of how to use TACL commands to suspend and activate processes, refer to the *Guardian User’s Guide* or the *TACL Reference Manual*

Suspending Your Own Process

To suspend your own process, you issue a PROCESS_SUSPEND_ procedure call without specifying any process. By default, the operating system selects your process for suspension:

```
CALL PROCESS_SUSPEND_;
```

The process then remains in the suspended state until reactivated by the RESUME command or the PROCESS_ACTIVATE_ procedure call from another process.

Suspending Other Processes

To suspend a process other than your own, you supply the PROCESS_SUSPEND_ procedure with the process handle of the process you want to suspend. The process handle is that returned by the PROCESS_LAUNCH_ procedure when the process was created. If you do not know the process handle, you can use the FILENAME_TO_PROCESSHANDLE_ procedure to find out the process handle; see Converting Between Process Handles and Process File Names later in this section.

The following example suspends a process identified by process handle:

```
ERROR := PROCESS_SUSPEND_(PROCESS^HANDLE);
IF ERROR <> 0 THEN ...
```

The process then remains in the suspended state until reactivated by the RESUME command or the PROCESS_ACTIVATE_ procedure.

If the process identified by PROCESS^HANDLE does not exist, then the PROCESS_SUSPEND_ procedure returns error 14.

If your process does not have the authority to suspend the identified process, then the PROCESS_SUSPEND_ procedure returns error 48. To have the authority to suspend a process, your process must either have the same process access ID as the process you want to suspend, be the group manager of that process access ID, or have the process access ID of the super ID user.

Activating Another Process

To activate a suspended process, supply the PROCESS_ACTIVATE_ procedure with the process handle of the process you want to activate.

```
ERROR := PROCESS_ACTIVATE_(PROCESS^HANDLE);
IF ERROR <> 0 THEN ...
```

The process then remains in the runnable state until suspended again by the SUSPEND command or the PROCESS_SUSPEND_ procedure.

If the process identified by PROCESS^HANDLE does not exist, then the PROCESS_ACTIVATE_ procedure returns error 14.

If your process does not have the authority to activate the identified process, then the PROCESS_ACTIVATE_ procedure returns error 48. To have the authority to activate a process, your process must either have the same process access ID as the process you want to activate, be the group manager of that process access ID, or have the process access ID of the super ID user.

Getting and Setting Process Information

You can use the PROCESS_GETINFO_, PROCESS_GETINFOLIST_, PROCESS_GETPAIRINFO_, and PROCESSHANDLE_GETMINE_ procedures to retrieve information about processes. The PROCESS_SETINFO_ and PROCESS_SETSTRINGINFO_ procedures enable you to set process information.

This subsection provides examples of how to use the above procedures to retrieve critical information. For complete details, refer to the *Guardian Procedure Calls Reference Manual*.

Getting Process Information

To retrieve information about existing processes, you can use the PROCESS_GETINFO_, PROCESS_GETINFOLIST_, PROCESS_GETPAIRINFO_, or PROCESSHANDLE_GETMINE_ procedure. PROCESS_GETINFO_ is convenient for retrieving specific information about a specific process, such as:

- The name of the home terminal
- The creator access ID and process access ID

- The process handle
- The process descriptor
- Information about related processes, such as the process handle of the job ancestor, the job ID, and the process handle of the mom process
- The name and length of the program file
- The name and length of the swap file
- The execution priority
- Process timing information (see Section 18, Managing Time)

PROCESS_GETINFOLIST_ provides lists of more detailed information about a specific process or about a list of processes that satisfy specified search criteria.

PROCESS_GETPAIRINFO_ provides information about a named process or process pair, including:

- The process handles of the primary and backup processes
- The process handle of the ancestor process
- The name of a process identified by process handle
- Lists of process names identified by a search string

PROCESSHANDLE_GETMINE_ provides an efficient way of obtaining the process handle of the calling process.

Using the PROCESS_GETINFO_ Procedure

Some examples of common uses of the PROCESS_GETINFO_ procedure follow. The first example returns the home terminal name for the process. The returned name is suitable for supplying to the FILE_OPEN_ procedure to open the terminal:

```
STRING  HOME^TERM[0:ZSYS^VAL^LEN^FILENAME - 1]
 .
 .
ERROR := PROCESS_GETINFO_(!process^handle!,
                          !file^name:maxlen!,
                          !file^name^len!,
                          !priority!,
                          !moms^processhandle!,
                          HOME^TERM:ZSYS^VAL^LEN^FILENAME,
                          HOME^TERM^LENGTH);
IF ERROR <> 0 THEN ...
 .
 .
ERROR := FILE_OPEN_(HOME^TERM:HOME^TERM^LENGTH,F^NUM);
IF ERROR <> 0 THEN ...
```

The next example returns the creator access ID and process access ID of the process named $P2:

```
PROCESS^NAME ':=' "$P2" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @PROCESS^NAME;
ERROR := FILENAME_TO_PROCESSHANDLE_(PROCESS^NAME:NAME^LENGTH,
                                    PROCESS^HANDLE);
IF ERROR <> 0 THEN ...

ERROR := PROCESS_GETINFO_(PROCESS^HANDLE,
                          !file^name:maxlen!,
                          !file^name^len!,
                          !priority!,
                          !moms^processhandle!,
                          !home^term:maxlen!,
                          !home^term^len!,
                          !process^time!,
                          CREATOR^ACCESS^ID,
                          PROCESS^ACCESS^ID,
                          !gmoms^processhandle!,
                          !jobid!,
                          !program^file:maxlen!,
                          !program^len!,
                          !swap^file:maxlen!,
                          !swap^len!,
                          ERROR^DETAIL);

IF ERROR <> 0 THEN
BEGIN
   CASE ERROR OF
   2 -> ...           !parameter error, ERROR^DETAIL
                      ! contains parameter number in error

   3 -> ...           !bounds error, ERROR^DETAIL contains
                      ! contains parameter number in error

   4 -> ...           !process does not exist

   OTHERWISE -> ...   !other error
END;
```

In the example above, the FILENAME_TO_PROCESSHANDLE_ procedure returns the process handle of process $P2. This process handle is then supplied to the PROCESS_GETINFO_ procedure to refer to the desired process.

Getting List Information About One Process

To use the PROCESS_GETINFOLIST_ procedure to get list information about one process, you supply the procedure with an identifier of the process you want detailed information about. The identifier can be the process handle or a combination of the node name, CPU number, and PIN.

The following example returns this information about the process specified in the *`processhandle`* parameter:

- Creator access ID
- Process access ID
- Home terminal name
- Process subtype
- Process execution time in microseconds
- Process state

```
! Instead of declaring the following literals, the program
! could also use the ones already declared in the
! PROCESS^ITEMCODES section of ZSYSDEFS.ZSYSTAL.

LITERAL CREATOR^ACCESS^ID^ATTR = 1,
        PROCESS^ACCESS^ID^ATTR = 2,
        HOME^TERM^NAME^ATTR    = 5,
        PROCESS^SUBTYPE^ATTR   = 8,
        PROCESS^TIME^ATTR      = 30,
        PROCESS^STATE^ATTR     = 32;
  .
  .
RETURN^ATTRIBUTE^LIST ':=' [CREATOR^ACCESS^ID^ATTR,
                            PROCESS^ACCESS^ID^ATTR,
                            HOME^TERM^NAME^ATTR,
                            PROCESS^SUBTYPE^ATTR,
                            PROCESS^TIME^ATTR,
                            PROCESS^STATE^ATTR];
RETURN^ATTRIBUTE^COUNT := 6;
RETURN^VALUES^MAXLEN := 2048;
ERROR := PROCESS_GETINFOLIST_(!cpu!,
                              !pin!,
                              !nodename:length!,
                              PROCESS^HANDLE,
                              RETURN^ATTRIBUTE^LIST,
                              RETURN^ATTRIBUTE^COUNT,
                              RETURN^VALUES^LIST,
                              RETURN^VALUES^MAXLEN,
                              RETURN^VALUES^LEN);

CREATOR^ACCESS^ID^VALUE := RETURN^VALUES^LIST[0];
PROCESS^ACCESS^ID^VALUE := RETURN^VALUES^LIST[1];
  .
  .
```

In the example above, RETURN^ATTRIBUTE^LIST contains a list of integer values that specify the information you want returned. The procedure returns the information in one array: RETURN^VALUES^LIST. The attribute values are returned in the order in which they were requested in RETURN^ATTRIBUTES^LIST.

The attribute values returned by PROCESS_GETINFOLIST_ can be of any valid data type. For variable-length strings (such as the home terminal name), the procedure first returns the number of bytes in the string, then the string itself. See the *Guardian Procedure Calls Reference Manual* for a discussion of all possible attributes and their data types.

Getting List Information About Multiple Processes

You can retrieve information about multiple processes from the PROCESS_GETINFOLIST_ procedure by supplying the procedure with search criteria information. You might, for example, want to retrieve information about all processes of a given priority, all processes with a specified creator access ID, or all processes started from a specified terminal.

The following example returns information about all processes with the same process access ID as the current process. The example uses the PROCESS_GETINFO_ procedure to determine the current process access ID and then supplies this value as the search criterion for the PROCESS_GETINFOLIST_ procedure.

```
CALL PROCESS_GETINFO_(!process^handle!,
                      !file^name:maxlen!,
                      !file^name^len!,
                      !priority!,
                      !moms^processhandle!,
                      !hometerm:maxlen!,
                      !hometerm^len!,
                      !process^time!,
                      !creator^access^id!,
                      PROCESS^ACCESS^ID);

PIN := 255;
RETURN^ATTRIBUTE^LIST ':=' (CREATOR^ACCESS^ID^ATTR
                            PROCESS^ACCESS^ID^ATTR,
                            HOME^TERM^NAME^ATTR,
                            PROCESS^SUBTYPE^ATTR,
                            PROCESS^TIME^ATTR,
                            PROCESS^STATE^ATTR);
RETURN^ATTRIBUTE^COUNT := 6;
RETURN^VALUES^MAXLEN := 2048;
SEARCH^OPTION = 2;
SEARCH^ATTRIBUTE^LIST := PROCESS^ACCESS^ID^ATTR;
SEARCH^ATTRIBUTE^COUNT := 1;
SEARCH^VALUES^LIST := PROCESS^ACCESS^ID;
SEARCH^VALUES^LEN := 1;
```

```
ERROR := PROCESS_GETINFOLIST_(!cpu!,
                              PIN,
                              !nodename:length!,
                              !process^handle!,
                              RETURN^ATTRIBUTE^LIST,
                              RETURN^ATTRIBUTE^COUNT,
                              RETURN^VALUES^LIST,
                              RETURN^VALUES^MAXLEN,
                              RETURN^VALUES^LEN,
                              ERROR^DETAIL,
                              SEARCH^OPTION,
                              SEARCH^ATTRIBUTE^LIST,
                              SEARCH^ATTRIBUTE^COUNT,
                              SEARCH^VALUES^LIST,
                              SEARCH^VALUES^LEN);
```

The *search-option* parameter, when set to 2, causes the procedure to search all processes with a PIN greater than or equal to the PIN specified in the *pin* parameter. In this example, the search is therefore restricted to all processes running at a high PIN (256 or greater).

Information for all processes identified in the search is returned in the *ret-values-list* parameter. If the information for the first matched process occupies *n* words, then information for the second process starts at *ret-values-list*[*n*].

Using the PROCESS_GETPAIRINFO_ Procedure

Some examples of typical uses of the PROCESS_GETPAIRINFO_ procedure follow. The first example returns the process handle and name of the ancestor process of the calling process:

```
INT ANCESTOR^PROCESS^HANDLE[0:ZSYS^VAL^PHANDLE^WLEN - 1];

ANC^MAXLEN := ZSYS^VAL^LEN^FILENAME;
ERROR := PROCESS_GETPAIRINFO_(!process^handle!,
                              !pair:maxlen!,
                              !pair^length!,
                              !primary^processhandle!,
                              !backup^processhandle!,
                              !search^index!,
                              ANCESTOR^PROCESS^HANDLE,
                              !nodename:length!,
                              !options!,
                              ANCESTOR^NAME:ANC^MAXLEN,
                              ANCESTOR^NAMELEN);
```

The next example uses the search index to find all processes on the node \CENTRAL. The setting of the *options* parameter includes I/O processes in the list.

```
MAXLEN := ZSYS^VAL^LEN^FILENAME^D00;
SEARCH^INDEX := 0D;
NODE^NAME ':=' "\CENTRAL" -> @S^PTR;
NODENAME^LENGTH := @S^PTR '-' @NODE^NAME;
```

```
OPTIONS := 1;

DO
BEGIN
   ERROR := PROCESS_GETPAIRINFO_(!process^handle!,
                                 PROCESS^NAME:MAXLEN,
                                 NAME^LENGTH,
                                 !primary^processhandle!,
                                 !backup^processhandle!,
                                 SEARCH^INDEX,
                                 !ancestor^processhandle!,
                                 NODE^NAME:NODENAME^LENGTH,
                                 OPTIONS);
   CASE ERROR OF
   BEGIN
      2 -> BEGIN
         ! Process parameter error
           .
           .
      END;
      3 -> BEGIN
         ! Process bounds error
           .
           .
      END;
      10 -> BEGIN
         ! Process error: unable to communicate with node
           .
           .
      END;
      OTHERWISE
      BEGIN
         IF ERROR <> 8 THEN
         BEGIN
            !  Process the name returned in NAME^LENGTH
               .
               .
      END;
   END;
END
UNTIL ERROR = 8;
```

Setting the *search-index* parameter to 0D causes the PROCESS_GETPAIRINFO_ procedure to search for a process name and return that name in PROCESS^NAME. The next time through the loop, it returns the next process name, and so on, until it has listed all process names on the specified node. The procedure returns an error value of 8 when it has completed the search.

Using the PROCESSHANDLE_GETMINE_ Procedure

A process can retrieve its own process handle by calling the PROCESSHANDLE_GETMINE_ procedure. While you can achieve the same result by passing a null process handle to the PROCESS_GETINFO_ procedure, PROCESSHANDLE_GETMINE_ performs this task more efficiently and without the need to initialize the process handle:

```
INT   .MYPHANDLE[0:ZSYS^VAL^PHANDLE^WLEN - 1];
 .
 .
ERROR := PROCESSHANDLE_GETMINE_(MYPHANDLE);
```

Setting Process Information

In addition to setting attribute values when creating a process, you can change the attribute values of existing processes using either the PROCESS_SETINFO_ or PROCESS_SETSTRINGINFO_ procedures. This subsection shows some examples of how to use these procedures. See the *Guardian Procedure Calls Reference Manual* for complete details on these procedure calls.

Setting Nonstring Process Attributes

To change a nonstring attribute of an existing process, use the PROCESS_SETINFO_ procedure. This procedure allows you to change the following attributes and optionally return the old value:

- Process priority.
- The process handle in the mom field in the PCB. Changing this value causes the Process deletion message (system message -101) to be sent to the new mom when the process terminates. This feature is useful only for unnamed process pairs.
- The process file security for the process. This value determines the security used for any file-creation attempt by the process following the call to PROCESS_SETINFO_. You can use this option only on your own process.
- The *primary* attribute, which indicates whether the process is the primary or backup of a process pair. You can set this value only for the calling process.
- The *qualifier-info-available* attribute, which determines whether qualifier-name searches by the FILENAME_FIND_ procedure are valid. You can set this value only for the calling process. See Section 13, Manipulating File Names, for information about the FILENAME_FIND_ procedure.

To change the mom entry in the PCB or the priority of a process, your process must have either the same process access ID as the process you want to change, the group manager's process access ID, or the process access ID of the super ID user. The remaining attributes can be changed only in the current process.

For example, the following call to PROCESS_SETINFO_ sets the *qualifier-info-available* attribute:

```
LITERAL QUALIFIER^INFO^AVAILABLE = 49;
 .
 .
SET^ATTRIBUTE^CODE := QUALIFIER^INFO^AVAILABLE;
SET^VALUE := 1;
SET^VALUE^LEN := 1;
CALL PROCESS_SETINFO_(!process^handle!,
                      !specifier!,
                      SET^ATTRIBUTE^CODE,
                      SET^VALUE,
                      SET^VALUE^LEN);
```

The *set-attr-code* parameter specifies the attribute to set; the *set-value* parameter specifies the new value of the attribute.

Setting String Process Attributes

To set process attributes that are strings, you use the PROCESS_SETSTRINGINFO_ procedure. The only attribute that is currently settable using this procedure is the home terminal name.

To set the home terminal name of a process, you supply the PROCESS_SETSTRINGINFO_ procedure with the process handle of the process whose home terminal you wish to change, the attribute code (5 for the home terminal), the new terminal name, and the name length. You can use the ZSYS^VAL^PINF^HOMETERM literal from the ZSYSTAL file to specify the attribute code.

The following example sets the home terminal for the current process to $TERM1. Note that for the current process, it is not necessary to provide the process handle.

```
SET^ATTRIBUTE^CODE := ZSYS^VAL^PINF^HOMETERM;
SET^ATTRIBUTE^VALUE ':=' "$TERM1" -> @S^PTR;
VALUE^LENGTH := @S^PTR '-' @SET^ATTRIBUTE^VALUE;
CALL PROCESS_SETSTRINGINFO_(
                           !process^handle!,
                           !specifier!,
                           SET^ATTRIBUTE^CODE,
                           SET^ATTRIBUTE^VALUE:VALUE^LENGTH);
```

To change the home terminal of a process, your process must have either the same process access ID as the process you want to change, the group manager’s process access ID, or the process access ID of the super ID user.

Manipulating Process Identifiers

This subsection describes system procedures that manipulate process handles. It discusses how to use the PROCESSHANDLE_DECOMPOSE_ procedure to retrieve information from a process handle as well as how to use the

FILENAME_TO_PROCESSHANDLE_ and PROCESSHANDLE_TO_FILENAME_ procedures to convert between process handles and process file names.

Retrieving Information From a Process Handle

To retrieve information from a process handle, use the PROCESSHANDLE_DECOMPOSE_ procedure. This procedure allows you to extract the following information:

- CPU number (*cpu* parameter)
- PIN (*pin* parameter)
- Node number (*nodenumber* parameter)
- Node name (*nodename* parameter) and the node-name length (*nlen* parameter)
- Process name (*procname* parameter) and the process-name length (*plen* parameter)
- Verification sequence number (*seqno* parameter)

To use the PROCESSHANDLE_DECOMPOSE_ procedure, you need to supply the process handle that you want to take apart and return variables for the information you need.

The following example retrieves all the above information from a process handle:

```
MAX^NODENAME^LEN := 8;
MAX^PNAME^LEN := 5;
ERROR := PROCESSHANDLE_DECOMPOSE_(PROCESS^HANDLE,
                                  CPU,
                                  PIN,
                                  NODE^NUMBER,
                                  NODENAME:MAX^NODENAME^LEN,
                                  NODENAME^LENGTH,
                                  PROCESS^NAME:MAX^PNAME^LEN,
                                  PROCESS^NAME^LENGTH,
                                  SEQUENCE^NUMBER);
```

Converting Between Process Handles and Process File Names

The following paragraphs describe how to use system procedure calls to convert named or unnamed process file names into process handles and how to convert process handles into process file names.

For a description of the syntax for a process file name, refer to Section 2, Using the File System.

Converting a Process File Name Into a Process Handle

To convert a process file name into a process handle, use the FILENAME_TO_PROCESSHANDLE_ procedure. This procedure works for named and unnamed process file names. For a named process, the operating system looks up the process handle in the DCT, and it returns error 14 (device does not exist) if the process name does not exist.

For unnamed processes, the operating system does not check for the existence of the process. It creates a process handle from the information provided in the process file name.

To use the FILENAME_TO_PROCESSHANDLE_ procedure, you must supply the procedure with the process file name and a return variable for the process handle. The following example converts a named process file name into a process handle:

```
PROCESS^NAME ':=' "$SRV1" -> @S^PTR;
NAME^LENGTH := @S^PTR '-' @PROCESS^NAME;
ERROR := FILENAME_TO_PROCESSHANDLE_(PROCESS^NAME:NAME^LENGTH,
                                    PROCESS^HANDLE);
IF ERROR = 14 THEN
BEGIN
   SBUFFER ':=' "No Such Process " -> @S^PTR;
   WCOUNT := @S^PTR '-' @SBUFFER;
   CALL WRITEX(TERM^NUM,SBUFFER,WCOUNT);
END;
```

Converting a Process Handle Into a Process File Name

To convert a process handle into a process file name, use the PROCESSHANDLE_TO_FILENAME_ procedure.

When using the PROCESSHANDLE_TO_FILENAME_ procedure, you must supply the process handle. The procedure returns the process file name in the *name* parameter and the name length in the *namelen* parameter.

If the process handle refers to a named process, then the procedure looks up the process file name in the DCT. (For a process pair, the procedure returns the current primary process.) If the process handle refers to an unnamed process, then the procedure returns the unnamed process file name using information contained in the process handle itself.

The PROCESSHANDLE_TO_FILENAME_ procedure returns the fully qualified process name (including the node name). The sequence number is optional. By setting bit 15 of the options parameter, you suppress the sequence number.

The following example converts a process handle into a fully qualified process file name, without the sequence number:

```
LITERAL MAXLEN   = ZSYS^VAL^LEN^FILENAME,
        NO^SEQNO = 1,
        SEQNO    = 0;
  .
  .
OPTIONS := NO^SEQNO;
CALL PROCESSHANDLE_TO_FILENAME_(PROCESS^HANDLE,
                                NAME:MAXLEN,
                                                 NAME^LENGTH,
                                                 OPTIONS);
```

Converting a Process Handle Into a Process String

To convert a process handle into a string, use the PROCESSHANDLE_TO_STRING_ procedure. This procedure is useful for producing a more readable output than that produced by the PROCESSHANDLE_TO_FILENAME_ procedure, particularly for unnamed processes. For example, a process file name returned by PROCESSHANDLE_TO_FILENAME_ could be "$:2:137:987654321." The equivalent output from PROCESSHANDLE_TO_STRING_ would be "2,137."

To use the PROCESSHANDLE_TO_STRING_ procedure, you must supply the process handle in the *`processhandle`* parameter and return variables for the string and string length in the *`pstring`* and *`pstringlen`* parameters.

The following example converts a process handle into a string:

```
LITERAL MAXLEN = ZSYS^VAL^LEN^FILENAME;
  .
  .
CALL PROCESSHANDLE_TO_STRING_(PROCESS^HANDLE,
                              PROCESS^STRING:MAXLEN,
                              STRING^LENGTH);
```

The above example returns either the process name if one exists or the CPU and PIN if no name exists. The above example also places the system name at the beginning of the output string.

Controlling the IPU Affinity of Processes

In multi-core CPUs, a process runs in a specific IPU of the CPU. The IPU to which a process is assigned is referred to as the IPU affinity of the process. By default the IPU affinity can be dynamically changed by the process scheduler for load-balancing or responsiveness purposes. You can override this for a specific process, binding it to a specific IPU via the IPUAFFINITY_SET_ procedure. The current IPU affinity of a process can be obtained via the IPUAFFINITY_GET_ procedure.

The IPUAFFINITY_CONTROL_ procedure can be used to override the process scheduler controls more generally to turn off dynamic load-balancing on all soft-affinity processes (see the definition of "soft affinity" below) or on all DP2 processes in the specified CPU.

The binding between IPUs and processes can only be done after the process is created, as it is not a process creation option.

Note. Beginning with the J06.12 and H06.23 RVUs, there are two instances of the TSMSGIP process active on each CPU on quad-core (4-IPU) systems, both of which by default are assigned to IPU 0. However, prior to J06.14 and H06.25, only the first instance is active and available to do work.

The TSMSGIP process handles interrupts generated by inter-CPU Message System traffic. Starting with J06.14 and H06.25, you can reassign one of the TSMSGIP instances to another IPU using the IPUAFFINITY_SET_ procedure. This is beneficial for certain workloads with high levels of inter-CPU Message System traffic.

IPU Affinity Classes

A process has one of the following types of IPU affinity, known as its IPU affinity class:

- *Hard* - the process can only be run on a specific IPU and is not subject to any kind of movement.
- *Group* - only applies to DP2 process groups (the one to eight processes that compose a disk volume). The group as a whole can be moved from IPU to IPU but only as the whole group.
- *Dynamic* - only applies to system processes known as Interrupt Processes (IPs) and Auxiliary Processes (APs). All of these processes other than the TSMSGIP, TSCOMIP, and TSSTRIP can run on any IPU as selected by the low level software for optimum response time, and are not subject to user control of the IPU placement. The TSMSGIP, TSCOMIP, and TSSTRIP processes are the only dynamic affinity processes that are subject to user control of their IPU affinity.
- *Soft* - all user processes and any other processes which do not fall into one of the other categories.
- *Soft Bind* - Soft Affinity processes whose IPU affinity has been set via the IPUAFFINITY_SET_ procedure.

The IPU affinity class of a process can be obtained via PROCESS_GETINFOLIST_ attribute 136.

IPU Affinity Control

The IPUAFFINITY_SET_ procedure is used to bind a process to an IPU. Once set the process will only run on the associated IPU. This procedure can be used on all user processes and many system processes. In particular it can be used on the ServerNet Interrupt Processes (TSMSGIP, TSCOMIP, and TSSTRIP), $TMP and DP2 process groups. The procedure is also used to disassociate a process from an IPU, thereby allowing the process scheduler to control any subsequent IPU assignments of the process.

The IPUAFFINITY_GET_ procedure can be used to get the current IPU a process associated with, as well as an indicator if the process can be the target of an IPUAFFINITY_SET call.

The IPUAFFINITY_CONTROL_ procedure is used to control Process Scheduler characteristics. It can cause the process scheduler to stop scheduling (that is changing) all processes with soft affinity or all DP2 processes (group affinity processes). It can also be used to disassociate all bindings that were created via IPUAFFINITY_SET_.

It is very important to remember that using IPUAFFINITY_SET_ and IPUAFFINITY_CONTROL_ restrict the function of the process scheduler to keep the CPU (that is all its IPUs) in balance. Much care should be exercised when utilizing these functions as an inappropriate choice can cause severe performance problems.

17 Managing Memory

An Introduction to Memory-Management Procedures

The following system procedures are available for managing memory from your application:

ADDRESS_DELIMIT_	Returns information about a particular area of the user's logical address space, including the addresses of the first and last bytes in that area.
GETPOOL_PAGE	Obtains a block of memory from a buffer pool.
HEADROOM_ENSURE_	Checks to make sure enough memory has been allocated for a process's main and priv stacks, and allocates more memory as needed. Can be called only from a native process.
MOVEX	Transfers data from one selectable segment to another.
POOL_CHECK_	Returns memory pool error information.
POOL_DEFINE_	Defines the bounds of a memory pool in an extended data segment or in the user data segment (TNS processes) or globals area (TNS/R native processes).
POOL_GETINFO_	Returns information about a memory pool.
POOL_GETSPACE_	Obtains a block of storage from a memory pool.
POOL_GETSPACE_PAGE	Obtains a block of memory from a buffer pool.
POOL_PUTSPACE_	Returns a block of storage to a memory pool.
POOL_RESIZE_	Changes the size of an existing memory pool.
PROCESS_CREATE_	In addition to creating a process, specifies the size of the user data segment for a TNS process.
PROCESS_LAUNCH_	Same as PROCESS_CREATE_, but provides additional parameters for specifying attributes associated with native processes.
RESIZESEGMENT	Changes the size of an existing data segment.
SEGMENT_ALLOCATE_	Allocates virtual memory space to an extended data segment (a flat segment or a selectable segment).
SEGMENT_DEALLOCATE_	Deallocates an extended data segment.

SEGMENT_GETINFO_	Returns information about an allocated extended data segment. The information returned may include the size of the extended data segment or the name of the associated swap file.
SEGMENT_USE_	Makes a selectable segment current. The current selectable segment is the only selectable segment that your process can access. (A flat segment need not be made current; your process can access all allocated flat segments.)
SETMODE	Option 141 can be used to speed large transfers of data between an extended data segment and a file.

Managing the User Data Areas

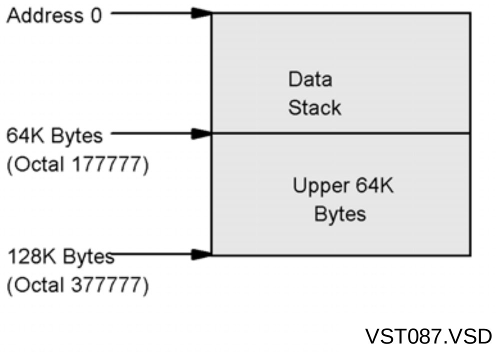
The structure of the areas of memory used by a process for user data differs for TNS processes and native processes.

Managing the TNS User Data Segment

In a TNS process, the user data segment provides up to 128K bytes of data storage for global variables, local variables, and return information for procedure calls. The lower 64K bytes of the user data segment are managed for you as a data stack by the operating system. To access the upper 64K bytes, you manage the data yourself if your program is written in TAL. Other languages use the upper 64K bytes for run-time environments. Refer to the *Common Run-Time Environment (CRE) Programmer's Guide* for details.

Figure 17-1 shows the user data segment.

Figure 17-1. The User Data Segment



It is possible for your TNS program to use more than 128K bytes of memory. See Using (Extended) Data Segments later in this section for details.

Specifying the Size of the TNS User Data Segment

You can specify the size of the user data segment to be any number of data pages of 2048 bytes each, up to a maximum of 128K bytes. You can set this size either by using the compiler or Binder program or when you call the PROCESS_CREATE_ or PROCESS_LAUNCH_ procedure.

To supply the size of the user data segment at compile/bind time, you set the compiler directive for the appropriate high-level language. For example, in TAL you would set the ?DATAPAGES directive. Refer to the appropriate compiler manual or your *Binder Manual* for details.

To specify the size of the user data segment using the PROCESS_CREATE_ or PROCESS_LAUNCH_ procedure, you must supply the `memory-pages` parameter. This parameter specifies a number of 2K-byte data pages.

The following example uses the PROCESS_CREATE_ procedure to create an unnamed process with six user data pages:

```
OBJFILE ':=' "PROGFILE" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR '-' @OBJFILE;
NAME^OPTION := 0;
MEMORY^PAGES := 6;
ERROR := PROCESS_CREATE_(OBJFILE:OBJFILENAME^LENGTH,
                         !library^file:lib^file^len!,
                         !swap^file:swap^file^len!,
                         !ext^swap^file:ext^swap^file^len!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         !name:length!,
                         PROCESS^DESCRIPTOR:MAXLEN,
                         PROCESS^DESCRIPTOR^LENGTH,
                         !nowait^tag!,
                         !hometerm:length!,
                         MEMORY^PAGES);
```

If you specify the number of data pages both as a compiler or Binder directive and as a parameter to the PROCESS_CREATE_ or PROCESS_LAUNCH_ procedure, the system uses the larger of the two values.

Using the Data Stack

The data stack occupies the first 64K bytes of the user data segment. It contains global data, local data for the main procedure, and dynamic local data for other TNS procedures.

Figure 17-2 shows how the data stack is used.

Figure 17-2(a) shows the stack before the main procedure starts to execute. Note that immediately after the global data is a zero entry called the dummy stack marker. The

S (stack) register points to the last entry in the stack; in this case, it points to the dummy stack marker.

When the main procedure executes, its local variables get added to the stack as shown in Figure 17-2(b). The S register points to the last location of the local variables.

When the main procedure calls another procedure (procedure 1 in Figure 17-2), the parameters for the new procedure are placed on the data stack followed by the contents of the P (program counter) register, E (environment) register, and L (local data address) register. Then the new procedure's local variables are placed on the stack (see Figure 17-2(c)). The S register points to the last location of the local variables of the new procedure.

Figure 17-2(d) shows what happens when procedure 1 calls procedure 2. Again the parameters for the new procedure are placed on the stack followed by the contents of the P register, E register, and L register. The S register is advanced to the last location of the local variables of procedure 2. Additional procedures can be nested in this way.

Figure 17-2. The Data Stack

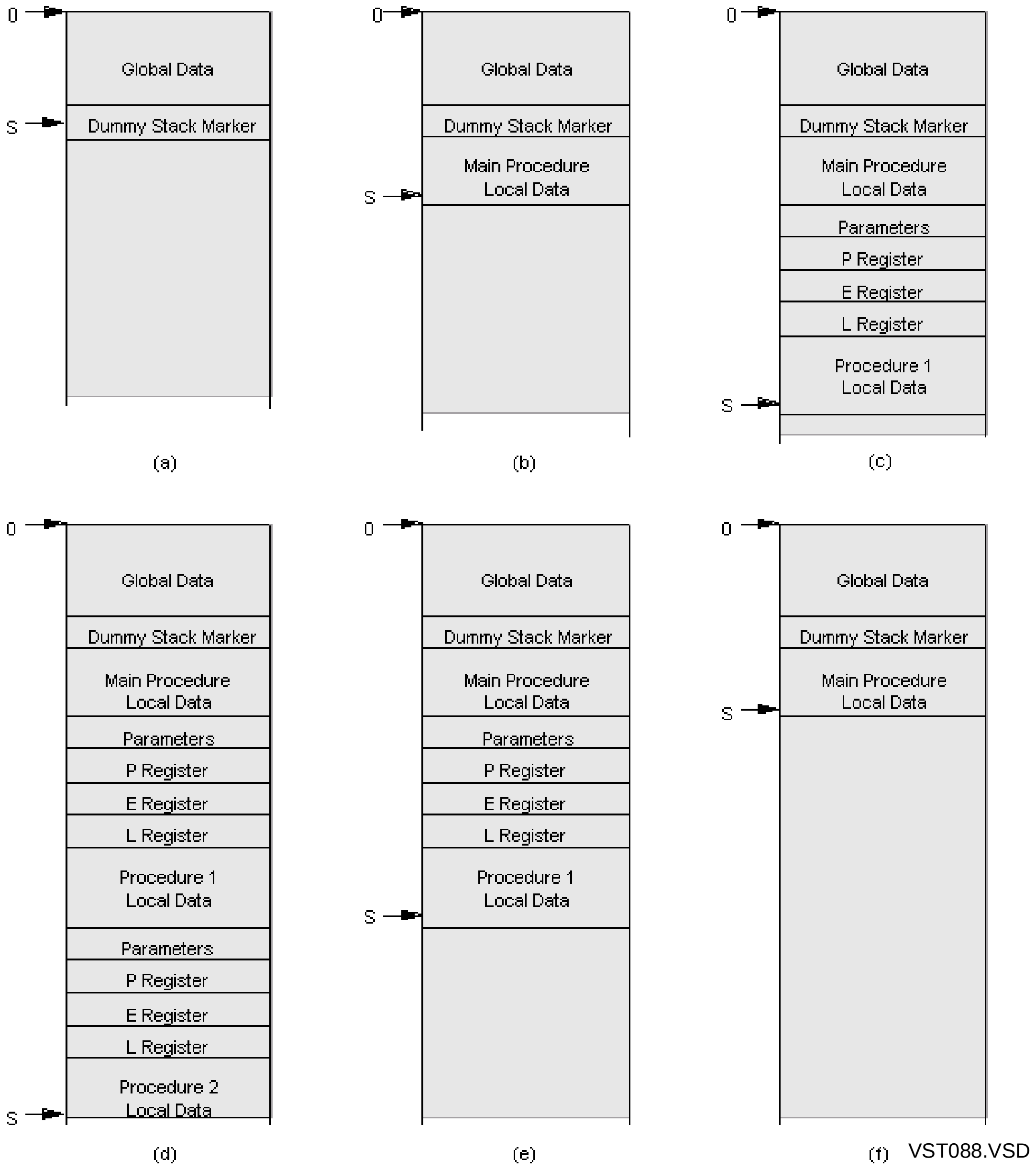

0
S
Global Data
Dummy Stack Marker
(a)
0
S
Global Data
Dummy Stack Marker
Main Procedure Local Data
(b)
0
S
Global Data
Dummy Stack Marker
Main Procedure Local Data
Parameters
P Register
E Register
L Register
Procedure 1 Local Data
(c)
0
S
Global Data
Dummy Stack Marker
Main Procedure Local Data
Parameters
P Register
E Register
L Register
Procedure 1 Local Data
Parameters
P Register
E Register
L Register
Procedure 2 Local Data
(d)
0
S
Global Data
Dummy Stack Marker
Main Procedure Local Data
Parameters
P Register
E Register
L Register
Procedure 1 Local Data
(e)
0
S
Global Data
Dummy Stack Marker
Main Procedure Local Data
(f)
VST088.VSD

Figure 17-2(e) shows what happens when procedure 2 returns. The process continues in the calling procedure at the program address immediately following the call to the procedure that just returned. It does this by restoring the saved values for the P register, E register, and L register. The S register moves back to the end of the local variables for procedure 1.

Figure 17-2(f) shows what happens when procedure 1 returns control to the main procedure. Again the P register, E register, and L register values are restored using the values saved on the stack so that processing continues at the address immediately following the call to the procedure that just returned. The S register points to the last location of the local variables of the main procedure.

The TNS user space also includes a main stack and a priv stack, which are used when a TNS procedure calls a TNS/R native procedure. When a nonprivileged native procedure is called, execution switches to the main stack. When a privileged native procedure is called, execution switches to the priv stack. The main and priv stacks are described later in this section under Managing the Native User Data Areas.

See the appropriate system description manual for a complete description of how the data stack, main stack, and priv stack work.

Using the Upper 64K Bytes of the Data Segment

Only the first 64K bytes of the user data segment can be used by the data stack. To access the upper 64K bytes, you must manage the space yourself using 16-bit memory pointers.

For example, in an application written in TAL, you could access an array starting at the beginning of the upper 64K bytes as follows:

```
INT .A := %100000;
  .
  .
X := A[4];
```

X is assigned the value of the fifth word of the upper 64K bytes.

Managing the Native User Data Areas

Data for TNS/R native processes is allocated in the following data segments:

- A globals-heap segment, containing global data and, for C and C++ programs, a user heap. This segment provides up to 128 megabytes of data storage.
- A main stack segment, containing the stack frames for nonprivileged procedure calls.
- A priv stack segment, containing the stack frames for privileged procedure calls.
- Zero or more SRL data segments used for global data owned by shared run-time libraries (SRLs). Each SRL owns an SRL data segment.
- A process file segment (PFS), used by the operating system for file buffers and other internal data.

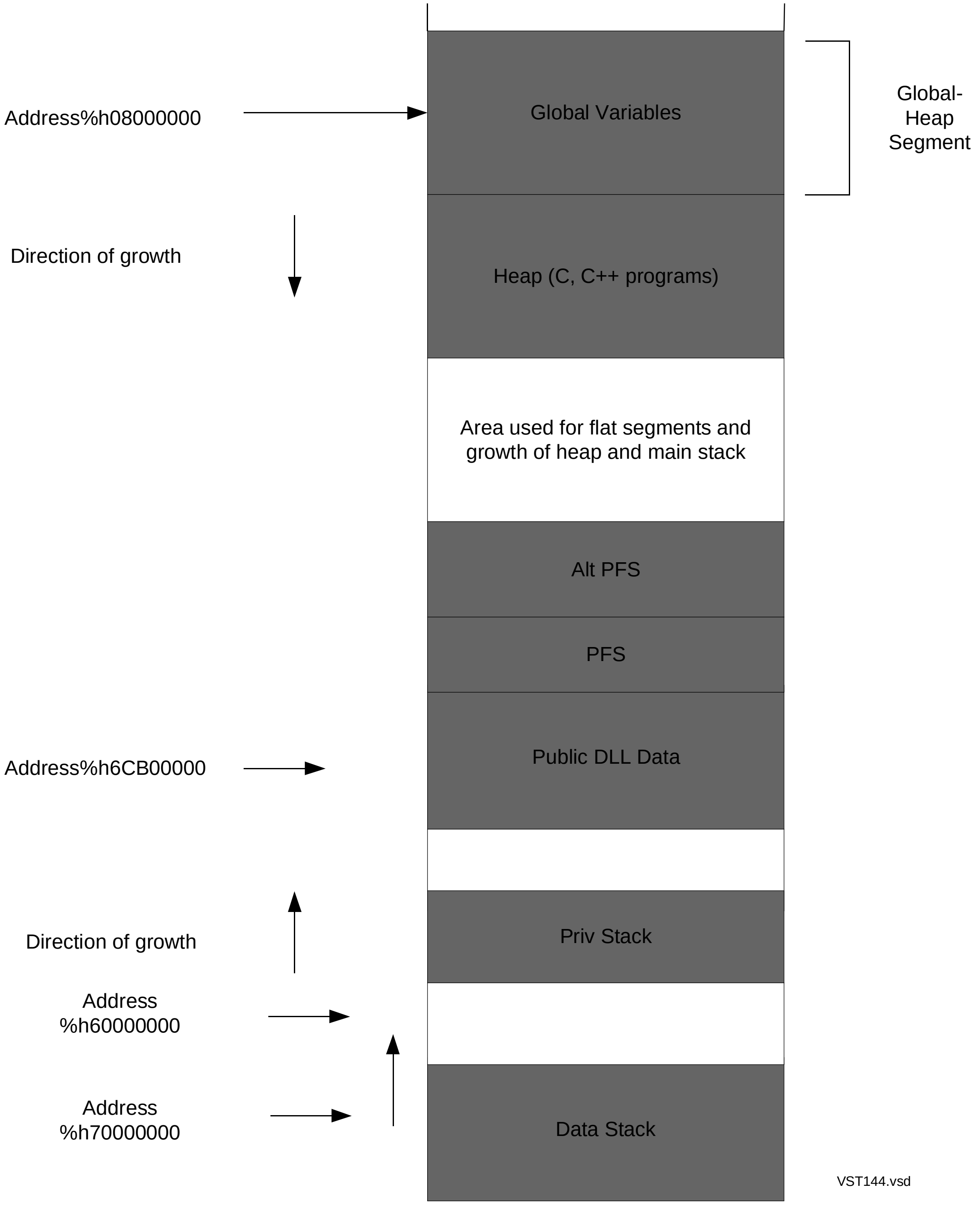
Data for a TNS/E native process is allocated into the following data segments:

- A globals-heap segment, containing program global data and, optionally, a heap
- A main memory stack for nonprivileged TNS/E native procedures
- A privileged memory stack for privileged procedures

- A main register stack engine (RSE) backing store for nonprivileged procedures
- A privileged RSE backing store for privileged procedures
- Zero or more DLL data segments
- Zero or more private DLL code segments
- A process file segment (PFS), used by the operating system
- Optional program-allocated extended data segments (selectable or flat segments)

Figure 17-3 shows the layout of the data segments for a native process.

Figure 17-3. Data Segments for TNS/R Native Processes


Address%h08000000
Direction of growth
Address%h6CB00000
Direction of growth
Address
%h60000000
Address
%h70000000
Global Variables
Heap (C, C++ programs)
Area used for flat segments and growth of heap and main stack
Alt PFS
PFS
Public DLL Data
Priv Stack
Data Stack
Global-
Heap
Segment
VST144.vsd

Note that the heap grows from lower to higher addresses, while the main and priv stacks grow from higher to lower addresses.

Also on both platforms, the heap can grow to the maximum size of the globals-heap segment less the size of the global data. On the TNS/R platform, the maximum globals-heap size is 1.1 GB. On the TNS/E platform, the maximum globals-heap size is 1536 MB.

If your program needs data areas in addition to the area provided by the globals-heap segment, you can allocate one or more flat segments or selectable segments, as described later in this section under Using (Extended) Data Segments.

How the Main and Priv Stacks Are Used

The main and priv stacks for a process are made up of stack frames, each of which contains the activation record for a procedure called during process execution: the main stack contains the stack frames for nonprivileged procedure calls, and the priv stack contains the stack frames for privileged procedure calls. When a nonprivileged process begins execution, a main stack and priv stack are created. Execution automatically switches to the priv stack when a privileged procedure is called and back to the main stack when that procedure finishes.

Contents of a stack frame include local variables, saved registers, and parameters to called procedures. The frame size is variable, depending on the number of registers, variables, and parameters. As shown in Figure 17-3, both the main stack and the priv stack begin at fixed addresses and grows from higher to lower addresses as additional space is needed. Use the HEADROOM_ENSURE procedure to ensure the priv stack has sufficient room to grow.

Figure 17-4 shows an example of a main stack for a native process.

When the main procedure starts to execute a stack frame is created for it and its local variables are added to the stack frame as shown in Figure 17-4(a). The sp (stack pointer) register points to the last (lowest-addressed) byte in the stack frame.

When the main procedure calls another procedure (procedure 1 in Figure 17-4), the main procedure places up to four parameters (in the TNS/R environment) or 8 parameters (in the TNS/E environment) into stacked registers and stores any additional parameters into an area within the stack frame known as the callout area. The instruction that transfers control to the called procedure also stores the return address into the ra (return address) register.

The called procedure (procedure 1) then does the following:

- Decrements the sp register to allocate its own stack frame, with room for local variables, saved registers, and enough callout space for parameters to be passed to any procedures it calls.
- Stores registers (including the return address and any of the saved registers that it will use) into its stack frame.
- Stores the parameter registers into their reserved locations in the main procedure’s callout area. This area is also referred to as the called procedure's callin area because the called procedure can access this area as part of its own stack frame.

(Note that if any of these operations are not needed, they might be eliminated by optimization.)

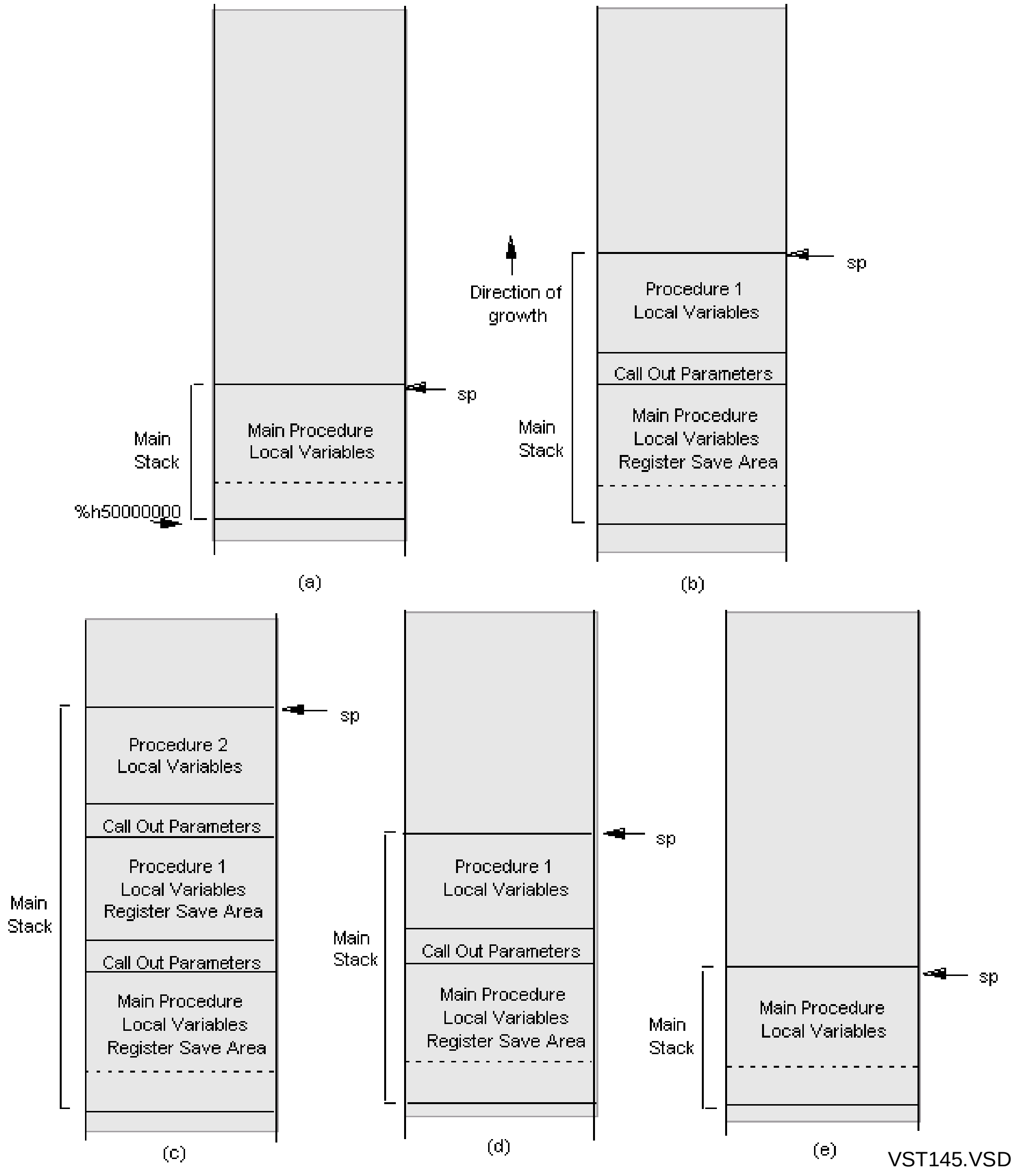
Figure 17-4(c) shows what happens when procedure 1 calls procedure 2. Again the calling procedure places up to four parameters into registers, stores any additional parameters into its callout area (which becomes the callin area for procedure 2), and stores the return address into the ra register. Additional procedures can be called in this way.

Figure 17-4(d) and 17-4(e) show how the main stack contracts as the called procedures return control to the calling procedures. When a called procedure returns control to a calling procedure, the called procedure:

- Restores the registers it saved, including the return address
- Increments the sp register to delete its stack frame
- Jumps to the location in the ra register

The sp register always points to the current stack tip; that is, the lowest-addressed byte in the stack frame of the currently active procedure.

Figure 17-4. Native Main Stack


Main Stack
Main Procedure Local Variables
sp
%h50000000
(a)
Direction of growth
Main Stack
Procedure 1 Local Variables
Call Out Parameters
Main Procedure Local Variables Register Save Area
sp
(b)
Main Stack
Procedure 2 Local Variables
Call Out Parameters
Procedure 1 Local Variables Register Save Area
Call Out Parameters
Main Procedure Local Variables Register Save Area
sp
(c)
Main Stack
Procedure 1 Local Variables
Call Out Parameters
Main Procedure Local Variables Register Save Area
sp
(d)
Main Stack
Main Procedure Local Variables
sp
(e)
VST145.VSD

Changing the Maximum Size of the Heap

The heap is managed by the Common Run-Time Environment (CRE). It is created at a system-defined initial size and is increased automatically as needed during process execution. For pre-G05 systems, the maximum allowable size of the heap defaults to the difference between 128 megabytes (the maximum combined size of the heap and globals area) and the size of the globals area (fixed at process creation time). In case of G05 and later systems, the default maximum allowable heap size of native process (the total of user heap, global variables, and flat segments) is 1120 MB and 1532 in all H-series and later RVUs.

You can change the maximum size of the heap through either the PROCESS_LAUNCH_ procedure or the native static loader (nld in the TNS/R environment and eld in the TNS/E environment). However, there is little reason to do so, since you can currently change it only to a smaller value. You might want to do this for debugging purposes; for example, to force a process to terminate with a heap overflow condition.

To set the maximum size for the heap through the PROCESS_LAUNCH_ procedure, specify the *`heap^max`* parameter in the procedure call or specify the *`heap_max`* option when invoking nld or eld. This sets the value of the HEAP^MAX process attribute, which defines the upper limit of the heap.

Changing the Maximum Size of the Main Stack

The size of the main stack increases automatically as needed during process execution up to a default limit of 1 megabyte in the TNS/R environment or 2 megabytes in the TNS/E environment. You can increase this limit up to a maximum of 32 megabytes either by calling the PROCESS_LAUNCH_ procedure or by using nld (in the TNS/R environment) or eld (in the TNS/E environment).

RISC and Itanium stack growth is as follows: (Note that TNS stacks grow upward.)

- RISC stacks grow downward (from higher to lower addresses)
- The Itanium RSE backing store grows upward
- Itanium memory stacks grow downward

On both TNS/R and TNS/E CPUs, the main stacks (main RISC stack on TNS/R, main memory stack on TNS/E) and the heap grow automatically as needed, to a maximum size. On the TNS/R platform, the default maximum stack size is one MB, and on the TNS/E platform, the default maximum stack size is two MB. You can increase the maximum stack size via an `eld` or PROCESS_LAUNCH_ parameter up to a limit of 32MB.

To set the maximum size for the main stack through the PROCESS_LAUNCH_ procedure, specify the *`mainstack^max`* parameter in the procedure call. This sets the value of the MAINSTACK^MAX attribute, which defines the maximum size, in bytes, of the main stack. The following example calls PROCESS_LAUNCH_ to create an unnamed process and set the upper bound of the main stack to 10 megabytes. The example sets the `mainstack^max` parameter to 10000000D.

```
STRING PROG_NAME[0:ZSYS^VAL^LEN^FILENAME-1];
INT .EXT ERROR_DETAIL,
         OUTPUT_LIST_LEN;
STRUCT OUTPUT_LIST(ZSYS^DDL^SMSG^PROCCREATE^DEF);
STRUCT PARAM_LIST(PROCESS_LAUNCH_PARAMS);
.
.
PARAM_LIST ':=' P_L_DEFAULT_PARMS_;  !initialize param struct
PROG_NAME ':=' "PROGFILE" -> @S^PTR;  !program file name
@PARAM_LIST.PROGRAM^NAME := $XADR(PROG_NAME);
PARAM_LIST.PROGRAM^NAME^LEN := $DBL(@S^PTR '-' @PROG_NAME);
PARAM_LIST.NAME^OPTIONS := ZSYS^VAL^PCREATOPT^NONAME;
PARAM_LIST.MAINSTACK^MAX := 10000000D;
ERROR := PROCESS_LAUNCH_(PARAM_LIST,
                         ERROR_DETAIL,
                         OUTPUT_LIST:$LEN(OUTPUT_LIST),
                         OUTPUT_LIST_LEN);
```

See Section 16, Creating and Managing Processes, for more information about calling PROCESS_LAUNCH_.

To set the main stack size using NLD, specify the `mainstack_max` option when invoking NLD. The following example uses NLD to set the maximum size of the main stack to 10 megabytes:

```
NLD -SET MAINSTACK_MAX 10000000
```

Increasing the Size of the Main and Priv Stacks

The main and priv stacks are assigned an initial size by the operating system at process creation time. The main stack then grows as needed during process execution, up to the value of the MAINSTACK^MAX process attribute. Because the main stack is managed for you, you seldom need to be concerned about the amount of stack space available at any given time. However, in cases where stack space is a concern, you can use the HEADROOM_ENSURE_ procedure to ensure that there is enough space, or “headroom,” in the stack for the needs of your process.

The priv stack is not adjusted automatically, but because it is used only by system library procedures, a nonprivileged procedure generally has no need to call HEADROOM_ENSURE_ to increase the size of the priv stack.

When calling HEADROOM_ENSURE_, you specify the number of bytes you think you will need in the stack. (The current size of the main stack is contained in the MAINSTACK^SIZE attribute, which you can check by calling the PROCESS_GETINFOLIST_ procedure.) HEADROOM_ENSURE_ then checks the size of the stack, attempts to enlarge it if necessary, and returns one of the following values:

- A value of zero indicates that either there was already enough room or the stack was enlarged to make enough room.

- A nonzero value indicates that there was not enough room and the stack could not be enlarged. The value also indicates the reason why the stack could not be enlarged (for example, the maximum allowable stack size would be exceeded, or memory or swap space could not be allocated).

If called from a nonprivileged procedure, HEADROOM_ENSURE_ operates on the main stack; if called from a privileged procedure, it operates on the priv stack.

You might want to use HEADROOM_ENSURE_ in the following cases:

- If your application uses unusually large local arrays or structs, or has many nested procedure calls, it can use HEADROOM_ENSURE_ to test whether the stack size is near the limit. You can then take appropriate action. (A stack overflow signal or trap results if a process attempts to increase the stack size beyond the limit.)
- If your application contains privileged procedures with unusually large stack requirements, it might need to call HEADROOM_ENSURE_, because the priv stack does not automatically increase as it is used.

Refer to the *Guardian Procedure Calls Reference Manual* for details on the HEADROOM_ENSURE_ procedure.

Reserving Swap Space

The SPACE^GUARANTEE process attribute establishes an amount of disk space to be used for all virtual memory requested by a process. This includes the globals-heap segment, the main stack, all the SRL data segments, and any flat segments allocated with default swapping. You can specify a value for this attribute in the PROCESS_LAUNCH_ procedure call or through NLD. Supplying this attribute allows a process to reserve enough memory and swap space when the process starts to ensure that execution will not be impacted by lack of swap space.

The value of the SPACE^INUSE process attribute indicates the amount of swap space currently available to a process. You can find out the current value of this attribute by calling the PROCESS_GETINFOLIST_ procedure and specifying the *`space_inuse`* parameter.

Note. Using the SPACE^GUARANTEE attribute to guarantee swap space is necessary only in unusual circumstances.

Checking the Bounds of Your Data Areas

You can use the ADDRESS_DELIMIT_ procedure to obtain the addresses of the first and last bytes of a particular area of your logical address space, such as your user data segment (TNS processes) or your globals-heap segment (native processes), and of your main and priv stacks. Knowing the bounds of your data area allows you, for example, to check parameter addresses.

You supply an address contained within the address area of interest, passing it to ADDRESS_DELIMIT_ in the value parameter *`address`*. You can also use the *`address-descriptor`* output parameter to obtain a set of flags that describe the area.

The following example can be run in both the native and TNS environments. In the TNS environment, the address of a local variable contained in the user data segment is passed to ADDRESS_DELIMIT_. The procedure returns the addresses of the first byte (`HIGH^ADDR`) and last byte (`LOW^ADDR`) of the user data segment, which are then used to determine its size.

In the TNS/R native environment, the address of a local variable contained in the main stack is passed to ADDRESS_DELIMIT_. The procedure returns the following:

- The address of the last byte in the main stack in `HIGH^ADDR`. This is equal to `MAINSTACK^ORIGIN - 1D`, where the `MAINSTACK^ORIGIN` process attribute indicates the starting byte address of the main stack.
- The current stack limit (the lowest-addressed byte allocated in the main stack segment) in `LOW^ADDR`. (This limit can change as the stack grows during execution.)

This example shows that the output addresses can be assigned either to a simple variable (`HIGH^ADDR`) or to a pointer variable (`LOW^ADDR`).

```
INT      LOCAL^VARIABLE;
STRING  .EXT LOW^ADDR;
INT(32)  HIGH^ADDR;
INT      ERROR,
         ERROR^DETAIL;
INT(32)  SIZE;
          .
          .
          .
ERROR := ADDRESS_DELIMIT_ ($XADR(LOCAL^VARIABLE),
                           @LOW_ADDR,
                           HIGH^ADDR,
                           ! address^descriptor ! ,
                           ! segment^ID ! ,
                           ERROR^DETAIL);

IF ERROR <> 0 THEN CALL ERROR^HANDLER;

SIZE := HIGH^ADDR - @LOW^ADDR + 1D;  ! size in bytes of
                                     ! user data segment
```

Using (Extended) Data Segments

The term "extended" is significant only in the context of TNS processes. The only nonextended data segment in user address space is the TNS user data segment, and there are no nonextended data segments in a native process.

When the user data segment (TNS processes) or globals area (native processes) does not provide enough data space for your process, you can make additional virtual memory available to the process. Virtual memory is allocated as one or more extended data segments. There are two types of extended data segments: flat segments and selectable segments. A flat segment can be any size up to 128 megabytes on G04.00 and all earlier G-series releases and all D-series releases; a selectable segment can be any size up to 127.5 megabytes on G04.00 and all earlier G-series releases and all D-series releases. A flat segment or a selectable segment can be up to 1120 megabytes on G05.00 and all subsequent G-series releases and 1536 megabytes on TNS/E. TNS and native processes can allocate both types of segments.

Throughout the remainder of this section, whenever a reference is made to an extended data segment, the information applies to both selectable segments and flat segments.

This subsection describes how to access extended data segments using Guardian procedures. It describes how to perform the following operations:

- Allocate an extended data segment using the SEGMENT_ALLOCATE_ procedure.
- Establish the current selectable segment using the SEGMENT_USE_ procedure.
- Determine the base address of a flat segment using the SEGMENT_USE_ procedure.
- Force a flat segment to be allocated at a specific address using the SEGMENT_ALLOCATE_ procedure.
- Pass data in an extended data segment to procedures that accept extended pointers, such as the READX and WRITEX procedures.
- Move data between selectable data segments using the MOVEX procedure.
- Move data between flat segments using assignment statements.
- Determine the size of an extended data segment using the SEGMENT_GETINFO_ procedure.
- Deallocate an extended data segment using the SEGMENT_DEALLOCATE_ procedure.
- Share an extended data segment.

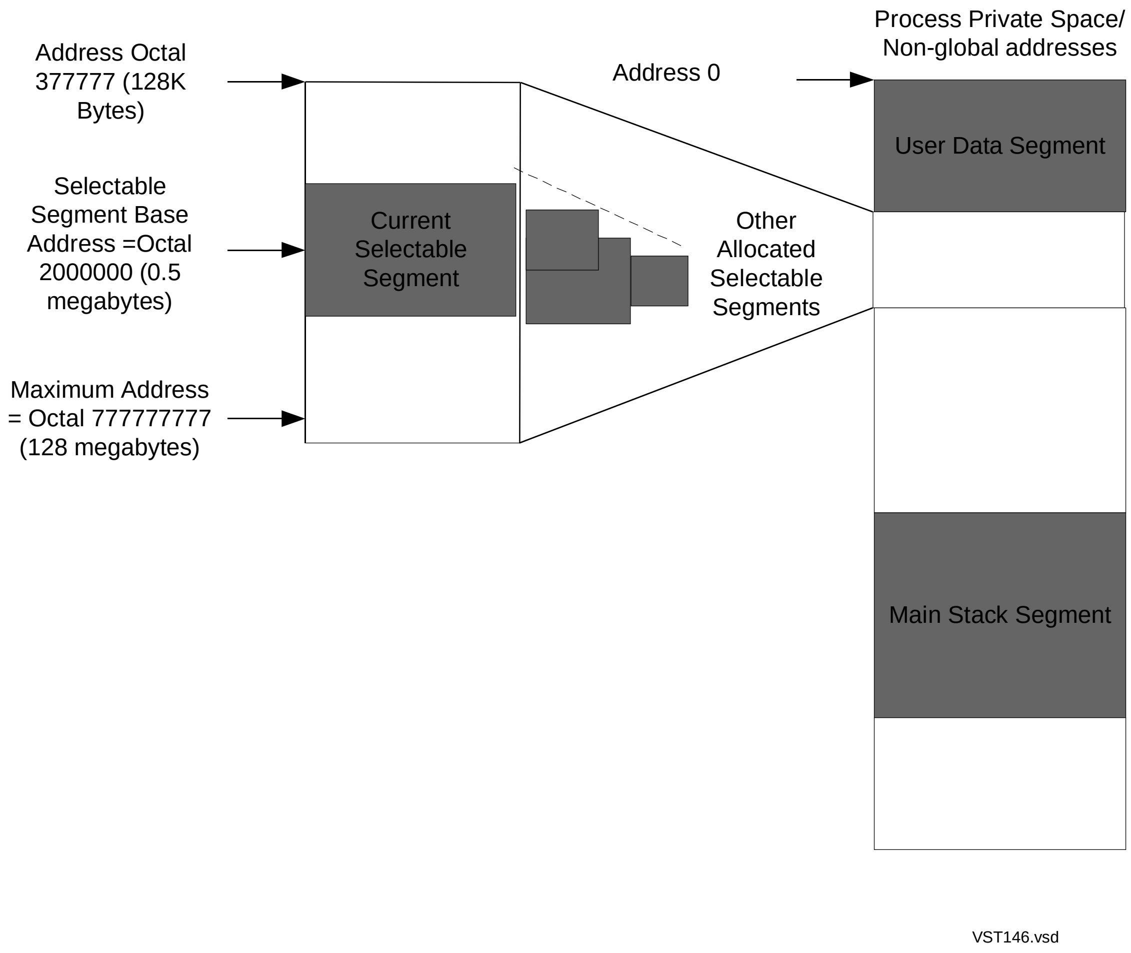
Overview of Selectable Segments

You can allocate space for multiple selectable segments; however, at most, one selectable segment is accessible at a time. It is called the current selectable segment.

A selectable segment starts at an address that is returned by the SEGMENT_ALLOCATE_ procedure. This address is %2000000 (%h00080000), and is the same for all selectable segments. Note that the selectable segment is not contiguous with the user data segment.

Figure 17-5 shows the location of selectable segments relative to the user data segment and main stack segment.

Figure 17-5. Relative Location of Selectable Segments


Address Octal 377777 (128K Bytes)
Selectable Segment Base Address =Octal 2000000 (0.5 megabytes)
Maximum Address = Octal 777777777 (128 megabytes)
Current Selectable Segment
Other Allocated Selectable Segments
Address 0
Process Private Space/ Non-global addresses
User Data Segment
Main Stack Segment
VST146.vsd

Overview of Flat Segments

The second type of extended data segment is the flat segment. You can allocate space for multiple flat segments, and all are accessible to the process that allocated them. Each flat segment is allocated at a different starting address on a 32-megabyte boundary (G04.00 and earlier G-series releases and all D-series releases; there is no boundary. In G05.00 and later G-series releases, only a limit of 1120 megabytes. In all H-series and J-series releases the limit is 1536 megabytes), and they do not overlap. All flat segments are accessible at the same time; unlike selectable segments, you need not make a flat segment the current segment in order to access it.

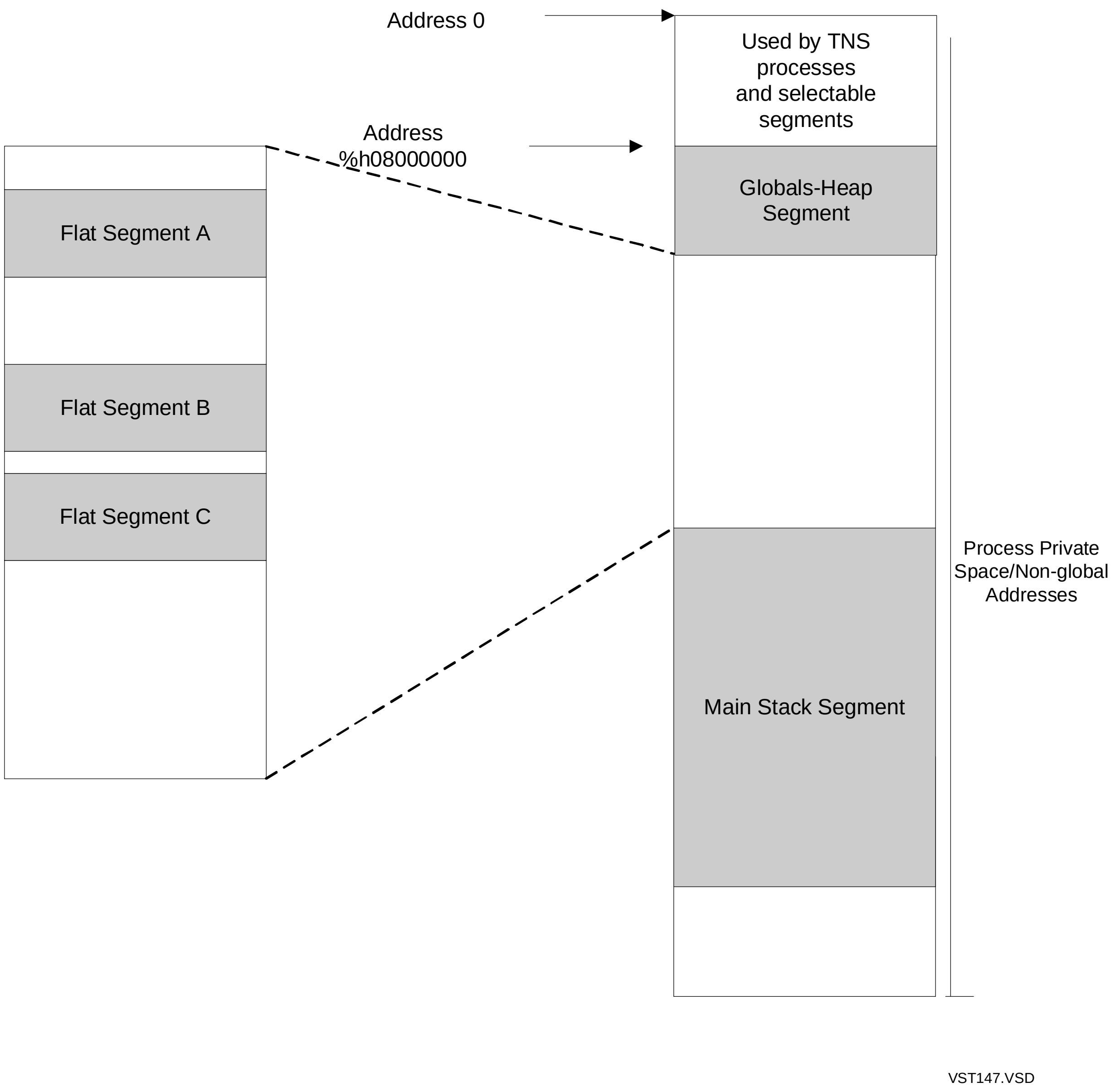
When you allocate a flat segment, you will generally allow the SEGMENT_ALLOCATE_ procedure to determine and return its starting address. You can optionally specify the starting address for a flat segment, but under most circumstances, this is not recommended.

For flat segments in a native mode process, the address space used for flat segments is also used for the heap (which is used by C / C++ applications). Flat segments (when allocated by Guardian) are assigned starting at the highest address and going downward. The heap starts at the lowest address and grows upward. This means, for native mode programs, the maximum segment size is not 1120 MB. It depends on how much heap space the program uses. At best, a native mode program has 1119 MB available for flat segments (it could have less available if the heap has grown to greater than 1 MB). An attempt to allocate an 1120 MB segment in a native program results in an error 15.

Figure 17-6 shows addressable areas of virtual memory for user data in a native process. Note that at any given time, a process can address all of the following:

- All flat segments
- One selectable segment (the current one)
- Global data
- The heap
- The main stack

Figure 17-6. Relative Location of Flat Segments


Address 0
Used by TNS processes and selectable segments
Address %h08000000
Globals-Heap Segment
Flat Segment A
Flat Segment B
Flat Segment C
Process Private Space/Non-global Addresses
Main Stack Segment
VST147.VSD

Which Type of Segment Should You Use?

Selectable segments are a carryover from earlier architectures. They will continue to be supported on newer systems. However, programs written for newer CPUs should use flat segments for the following reasons:

- Flat segments provide a performance advantage. Unlike selectable segments, all flat segments allocated within a process are accessible to the process at the same time. You need not make a special call to the SEGMENT_USE_ procedure to make the flat segment the current segment before accessing it. In addition, you can move data between flat segments by using assignment statements or move statements; use of the MOVEX procedure is not required. Depending on the number of SEGMENT_USE_ and MOVEX calls in your program, removing them can provide a significant performance enhancement.

- Flat segments provide access to more virtual memory and enable you to access areas of memory that were previously inaccessible.
- Flat segments are more convenient from a programming standpoint, because programs do not need calls to SEGMENT_USE_ or MOVEX.
- If you have more than one selectable segment, you should use flat segments to prevent performance degradation when switching between selectable segments, because only one selectable segment is visible at a time. Flat segments are always visible.
- TNS processes using the automatic extended data segment should avoid selecting other segments; see the following section.

You might still want to use selectable segments to simplify migration to newer systems or for programs to be executed on both older and newer systems.

Using Selectable Segments in TNS Processes

Many TNS processes use an automatic extended data segment, a selectable segment with ID 1024. It is created implicitly by the TNS C, Fortran and Pascal compilers when using the large-memory model (XMEM, often called the LARGE or WIDE model in TNS C). The TAL compiler also creates the automatic extended segment if any aggregate variables (structures and arrays) are declared with extended indirection (.EXT). By default, C places the heap, most global and static variables, and local aggregate variables in the extended segment; the default can be overridden through use of the `_lowmem` storage specifier. The automatic segment is selected when the process starts, and remains selected unless some other segment is selected.

HP recommends against explicitly calling SEGMENT_USE_ or USESEGMENT to select other selectable segments in TNS processes that use the automatic extended data segment. Explicit segment selection is possible, but fraught with difficulty: When the program selects some other segment, the automatic segment becomes invisible, so the heap and many variables become inaccessible. Any reference to one of these areas becomes a reference to the same address in the currently selected segment, leading to incorrect program behavior and likely data corruption. Therefore, the programmer must avoid making any such references while the other segment is selected. Also, the program must avoid calling any functions that depend on the heap. That set of functions is not explicitly documented, but includes many common ones, such as `printf()`. If explicit segment selection is necessary, limit the duration of the selection to as few statements as possible, and ensure that those statements refer only to local scalar variables or variables qualified by the `_lowmem` storage specifier; avoid calling most run-time library functions. Immediately reinstate the automatic segment by selecting the previously selected segment, using the ID returned via the *`old-segment-id`* parameter to SEGMENT_USE_.

The program can safely create or share a selectable segment without selecting it, instead using the `MOVEX` procedure to copy data between it and the normal program environment.

Accessing Data in Extended Data Segments

You can access data in flat segments and selectable segments by using Guardian procedures. In addition, you can access data in selectable segments by using extended indirect arrays from an application written in TAL or other languages supported by the TNS environment.

- To access data in an extended data segment using Guardian procedures, you must first allocate the segments you need using the SEGMENT_ALLOCATE_ procedure. If the segment is a selectable segment, you must then specify the segment you want to use by calling the SEGMENT_USE_ procedure. (If the segment is a flat segment, you need not call SEGMENT_USE_.) For all segments, this method lets you allocate as many extended data segments as you need up to a total of:
    - 480 megabytes on G04.00 and earlier G-series releases and all D-series releases
    - 1120 megabytes on G05.00 and subsequent G-series releases
    - 1536 megabytes on all H-series and J-series releases

  For selectable segments, you can use only one segment at a time. For flat segments, you can access any of the allocated segments.

- To access data in a selectable segment using extended indirect arrays, you can simply declare the array using the .EXT keyword. TAL automatically allocates an extended data segment for your program:

  ```
  INT .EXT MYDATA[0:99];
  ```

  This example declares a 100-word array.

  Extended indirect arrays, although easy to use, provide access to only one selectable segment.

  For a TNS C program using the Large or Wide model, or a COBOL, FORTRAN, or Pascal program, the compiler can also place data in a selectable data segment, which also typically contains the heap.

  The native compilers do not create selectable segments. In pTAL, the preceding example would place MYDATA in the globals segment or, for a local declaration, in the stack frame.

When accessing selectable segments in a TNS program, you should choose one method or the other. You should not mix the two methods.

For more details on how to use indirect extended arrays, see the *TACL Reference Manual* or the *pTAL Reference Manual*.

Attributes of Extended Data Segments

Normally, an extended data segment is private to the process that owns it, allows both read and write access, and is created with its intended size. However, you can create extended data segments with special properties that permit them to be:

- Extensible, allowing dynamic allocation of disk space to the swap file
- Shared with another process on the same CPU
- Read-only, to keep the contents from being altered, which permits sharing across CPUs

The following paragraphs describe the data segment attributes.

Extensible Data Segments

The system allocates extents to the swap file for an extensible data segment when needed. Initially, the swap file might have no extents assigned to it for a private extended data segment. If the data segment is to be shared, then one extent is initially assigned to the swap file.

Shared Data Segments

For processes that share data, you can use shared data segments. You specify sharing when allocating the segment. You can do so either by specifying the segment ID of the segment and the PIN of an existing process you want to share data with or by specifying the same swap file as an existing extended data segment. See Sharing an Extended Data Segment, later in this section for details on how to do this.

Because the part of the shared data segment that is in memory can contain written information that has not yet been copied to the swap file, you cannot share data segments across CPUs. All processes sharing the same data segment must run on the same CPU, unless the segment is read-only, as explained below.

Read-Only Data Segments

The contents of a read-only data segment cannot be altered and its data pages are not copied to the swap file. Being read-only makes it possible to share segment contents across CPUs.

Read-only segments cannot be extensible.

Allocating Extended Data Segments

To allocate an extended data segment, use the SEGMENT_ALLOCATE_ procedure.

Note. Most procedure calls with the “SEGMENT_” prefix return an `error-detail` parameter as well as a return code in the `error` variable.

Allocating a Selectable Segment

Using the SEGMENT_ALLOCATE_ procedure call, you can allocate one or more selectable segments. Each selectable segment can be as many as:

- 127.5 megabytes long on G04.00 and earlier G-series releases and all D-series releases
- 1120 megabytes long on G05.00 and all subsequent G-series releases
- 1536 megabytes long on all H-series and J-series releases

To allocate a selectable segment, you specify a segment ID, which is any number in the range 0 through 1023, in the *`segment-ID`* parameter of the SEGMENT_ALLOCATE_ procedure. You use the segment ID later when you make the segment current.

You also specify the size of the segment in bytes and a variable to receive the address of the segment base. The size and base address must be specified as 32-bit integers.

The following example allocates four selectable segments of 100,000 bytes each. These selectable segments are identified by segment IDs 0 through 3:

```
FOR I := 0 TO 3 DO
BEGIN
   SEGMENT^ID[I] := I;
   SIZE := 100000D;
   ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID[I],
                              SIZE,
                              !swap^file^name:length!,
                              ERROR^DETAIL,
                              !pin!,
                              !segment^type!,
                              BASE^ADDRESS[I]);
   IF ERROR <> 0 THEN CALL ERROR^HANDLER;
END;
```

The *`base-address`* output parameter returns the address of the start of the allocated segment. This address is the same for all selectable segments.

The preceding example also specifies the optional *`error-detail`* parameter. This parameter returns a value if the returned *`error`* parameter is nonzero. The *`error-detail`* parameter qualifies the *`error`* value.

Allocating a Flat Segment

Using the SEGMENT_ALLOCATE_ procedure call, you can allocate one or more flat segments. To allocate a flat segment, specify, at minimum, the following parameters in the SEGMENT_ALLOCATE_ procedure call:

- A unique segment ID, which is any number in the range 0 through 1023, in the *`segment-ID`* parameter.

- The size of the segment in bytes. The maximum size of a flat segment is 1120 megabytes in the TNS/R environment and 1536 megabytes in the TNS/E environment. In case of native mode C / C++ applications, the maximum size of a flat segment may not be 1120 MB since the address space used for flat segments is also used for the heap.
- The *`base-address`* parameter. You can specify this parameter as either an output parameter or an input parameter. If you specify it as an output parameter, the base address of the flat segment is determined internally and returned in the variable you specify. The base address value is different for each allocated segment. If you specify an address as an input value in the *`base-address`* parameter (not recommended), the segment is allocated using that address as a base address, if possible.
- The *`options`* parameter with bit 14 set to 1. This parameter tells the SEGMENT_ALLOCATE_ procedure to allocate a flat segment. The default setting of bit 14 is 0, which specifies a selectable segment. If you are specifying the *`base-address`* parameter as an input parameter, you must also set bit 15 to 1.

Note that for compatibility with legacy applications, a selectable segment is the default allocation on TNS/R systems. If you omit the *`options`* parameter, a selectable segment is allocated.

In most cases, you should specify the *`base-address`* parameter as an output parameter and allow the SEGMENT_ALLOCATE_ procedure to designate the starting address of the flat segment. In particular, library procedures that allocate flat segments should not specify a base address, because the allocation might be incompatible with other segments within the same process.

However, you might want to designate a starting address for a flat segment to ensure that the same address is used in a backup and primary process pair. The base address you specify:

- Must be within the address range in which flat segments can be allocated. (This address is subject to change.)
- Must not cause the allocated segment to overlap a flat segment that has already been allocated by the process.
- Must be a multiple of 128K bytes.

If you specify the *`base-address`* parameter as an input parameter, you must set bit 15 of the *`options`* parameter to 1.

If you specify a base address for a flat segment and that address range is not available, an error is returned.

Example of Allocating Extended Data Segments

The following example allocates four flat segments of 100,000 bytes each. These flat segments are identified by segment IDs 0 through 3:

```
OPTIONS := 0;
OPTIONS.<14> := 1;
FOR I := 0 TO 3 DO
BEGIN
   SEGMENT^ID[I] := I;
   SIZE := 100000D;
   ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID[I],
                              SIZE,
                              !swap^file^name:length!,
                              ERROR^DETAIL,
                              !pin!,
                              !segment^type!,
                              BASE^ADDRESS[I],
                              !max^size!,
                              OPTIONS);
   IF ERROR <> 0 THEN CALL ERROR^HANDLER;
END;
```

The *`base-address`* output parameter receives the starting address of each allocated segment.

The preceding example also supplies the optional *`error-detail`* parameter. This parameter returns a value if the returned *`error`* parameter is nonzero. The *`error-detail`* parameter provides more information about the error.

Managing Swap Space

Data pages in physical memory are regularly swapped to a disk file to release memory for other needs. By default, swap space for an extended data segment is handled by the Kernel-Managed Swap Facility (KMSF). (This is the preferred way to handle swap space for an extended data segment.) For each CPU, KMSF manages one or more swap files from which swap space is allocated for the processes in that CPU. For more information about KMSF, refer to the *Kernel-Managed Swap Facility (KMSF) Manual.*

Alternatively, you can specify that an extended data segment have its own swap file, which can be a temporary file or a permanent file. You specify a swap file by supplying the *`filename`*`:`*`filename-length`* parameter to the SEGMENT_ALLOCATE_ procedure. If you supply a file name that includes a file ID part, and if a file with that name does not already exist, then the system creates a permanent swap file with that name. You can also specify just the volume part of the file name, in which case the system creates a temporary swap file on the specified volume.

If you specify the name of an existing file to use as a swap file, you must have read access to the file if it is to be used for a read-only segment, or read/write access otherwise.

The difference between the temporary swap file and the permanent swap file is that when the extended data segment is later deallocated, the permanent swap file remains on disk and can be accessed after the segment is deallocated. It can be used, for example, to set the initial data in another segment that later uses it as a swap file; thus, it can serve as a record of a past state. In contrast, the data in a temporary swap file is lost once you deallocate the extended data segment.

The following example allocates a flat segment with a permanent swap file:

```
SEGMENT^ID := 4;
SIZE := 8000D;
OPTIONS.<14> := 1;
FILENAME ':=' "$PROGRAM.SWPFILES.MYPROG" -> @S^PTR;
FILENAME^LENGTH := @S^PTR '-' @FILENAME;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           SIZE,
                           FILENAME:FILENAME^LENGTH,
                           ERROR^DETAIL,
                           !pin!,
                           !segment^type!,
                           BASE^ADDRESS,
                           !max^size!,
                           OPTIONS);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

You must specify a file name if you intend to use the file-name method of segment sharing.

Specifying the Attributes of the Extended Data Segment

So far this section has described how to allocate extended data segments with default attributes: that is, private, read/write, and nonextensible. If you need segments that are shared with another process, are read-only, or are extensible, then you need to specify the *`segment-type`* parameter. The following example specifies an extensible data segment. The value for the *`segment-type`* parameter, ZSYS^VAL^SEGALLOCTYPE^EXTENSBL, is taken from the ZSYSTAL file:

```
SEGMENT^ID := 0;
SIZE := 8000D;
SEGMENT^TYPE := ZSYS^VAL^SEGALLOCTYPE^EXTENSBL;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           SIZE,
                           !swap^file^name:length!,
                           ERROR^DETAIL,
                           !pin!,
                           SEGMENT^TYPE,
                           BASE^ADDRESS);
IF ERROR <> 0 THEN ...
```

For a complete list of segment-type values and other details about the SEGMENT_ALLOCATE_ procedure, see the *Guardian Procedure Calls Reference Manual*.

Checking Whether an Extended Data Segment Is Selectable or Flat

Use the SEGMENT_GETINFO_ procedure to check whether a previously allocated extended data segment is a selectable segment or flat segment. Given the *segment-id* of an extended data segment, the SEGMENT_GETINFO_ procedure returns an option flag indicating whether the segment is a flat segment or selectable segment: if bit 9 of the *usage-flags* parameter is 1, the segment is a flat segment; if bit 9 is 0, the segment is a selectable segment. This check is useful when writing transportable programs for earlier systems.

Alternatively, you can check whether a segment is flat or selectable by testing the *base-address* value returned by SEGMENT_ALLOCATE_. If the value is %2000000 (%H00080000), the segment is a selectable segment; otherwise, the segment is a flat segment.

The following example checks the *usage-flags* parameter to determine the segment type of extended data segment 1:

```
SEGMENT^ID := 1;
ERROR := SEGMENT_GETINFO_(SEGMENT^ID,
                         SIZE,
                         !swap^file:maxlength!,
                         !filename^length!,
                         ERROR^DETAIL,
                         BASE^ADDRESS,
                         USAGE^FLAGS);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
IF USAGE^FLAGS.<9> = 1 THEN
<Processing for flat segment>
ELSE
<Processing for selectable segment>
```

Making a Selectable Segment Current

Before you can access an allocated selectable segment, you must make the selectable segment current by issuing a SEGMENT_USE_ procedure call. Your program can refer only to the current selectable segment, and only one selectable segment can be current at any time.

You specify the selectable segment you want to make current in the *segment-id* parameter of the SEGMENT_USE_ procedure call. This segment ID must be the same as the segment ID you supplied to the SEGMENT_ALLOCATE_ procedure. If the segment ID is invalid, the SEGMENT_USE_ procedure returns an error 4.

With successful completion of the SEGMENT_USE_ procedure call, the procedure returns the previous value of the current segment ID in the optional *old-segment-id* parameter. If no selectable segment was in use before the call to SEGMENT_USE_, the procedure returns -1.

You can also specify the optional *`base-address`* output parameter to return the address of the start of the allocated segment, along with the optional *`error-detail`* parameter to return a value giving more information about any nonzero *`error`* value.

The following example allocates four selectable segments and specifies that the first of these (segment ID 0) is to be the current selectable segment:

```
FOR I := 0 TO 3 DO
BEGIN
   SEGID[I] := I;
   SIZE := 100000D;
   ERROR := SEGMENT_ALLOCATE_(SEGID[I],
                              SIZE,
                              !swap^file:length!,
                              ERROR^DETAIL,
                              !pin!,
                              !segment^type!,
                              BASE^ADDRESS[I]);
   IF ERROR <> 0 THEN CALL ERROR^HANDLER;
END;

NEW^SEGMENT^ID := 0;
ERROR := SEGMENT_USE_(NEW^SEGMENT^ID,
                      OLD^SEGMENT^ID,
                      @SEG^PTR,
                      ERROR^DETAIL);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

After your program calls the SEGMENT_USE_ procedure, all references to selectable segments access the selectable segment that was specified in the SEGMENT_USE_ call. Although the other selectable segments cannot be accessed by your process (unless SEGMENT_USE_ is called), they remain allocated; data in the other segments is therefore retained. The MOVEX procedure can be used to copy data into or out of a selectable segment that is not currently in use.

Referencing Data in an Extended Data Segment

Once you have allocated an extended data segment and, if it is a selectable segment, made it current, you can refer directly to locations in that extended data segment. You do this by using extended pointers in TAL or pTAL.

An extended pointer is a 32-bit address of an extended data segment. (An extended pointer can also contain the address of a TNS user data segment storage location; see Using the Data Stack earlier in this section.)

You declare an extended pointer using the .EXT keyword. The following examples show extended pointers used to access selectable and flat segments.

Referencing Data in a Selectable Segment

Following are extended pointers for STRING, INT, and INT(32) data types. The pointers will be used to access a selectable segment.

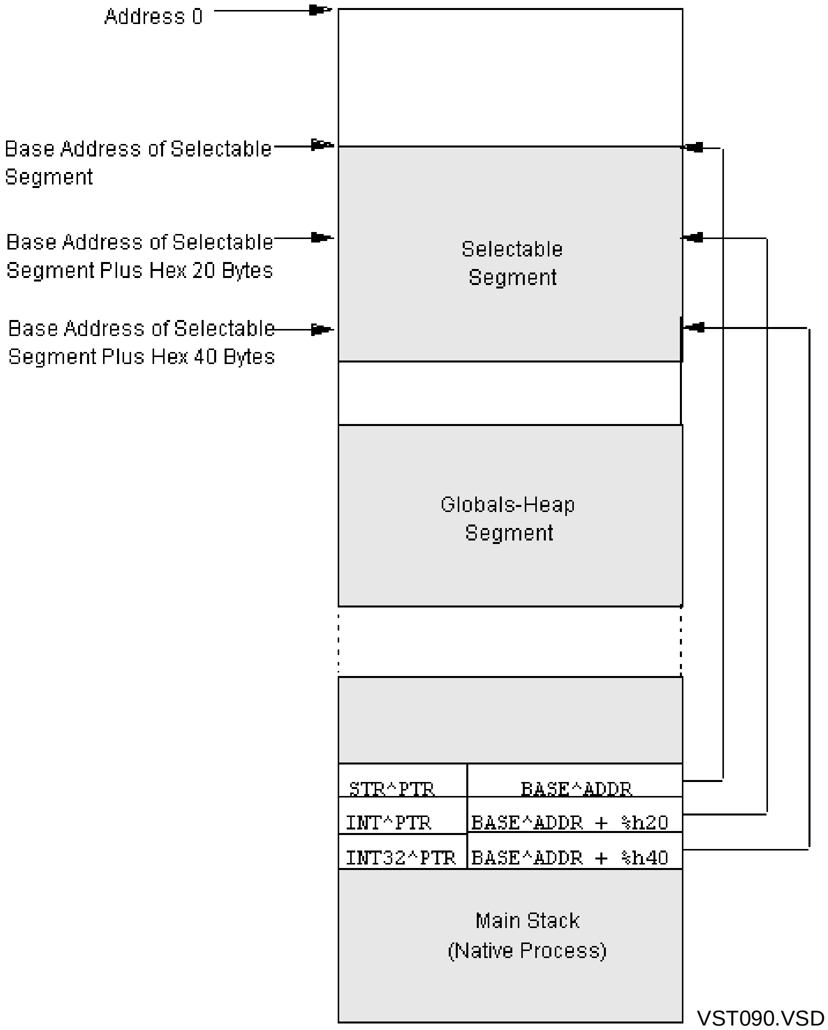
```
INT(32)  BASE^ADDR;
STRING  .EXT STR^PTR;
INT     .EXT INT^PTR;
INT(32) .EXT INT32^PTR;
```

```
@STR^PTR   := BASE^ADDR;
@INT^PTR   := BASE^ADDR + %H20%D;
@INT32^PTR := BASE^ADDR + %H40%D;
```

In the above statements, BASE^ADDR is the byte address of the first location in the selectable segment and was returned either by the call to SEGMENT_ALLOCATE_ that allocated this segment or by a call to the SEGMENT_GETINFO_ procedure.

Figure 17-7 shows the effect of the pointers declared above.

Figure 17-7. Referencing a Selectable Segment


Address 0
Base Address of Selectable Segment
Base Address of Selectable Segment Plus Hex 20 Bytes
Base Address of Selectable Segment Plus Hex 40 Bytes
Selectable Segment
Globals-Heap Segment
STR^PTR
BASE^ADDR
INT^PTR
BASE^ADDR + %h20
INT32^PTR
BASE^ADDR + %h40
Main Stack (Native Process)

VST090.VSD

Note

Pointers to selectable segment set up by:

```
@STR^PTR     := BASE^ADDR;
@INT^PTR     := BASE^ADDR + %H20%D;
@INT32^PTR   := BASE^ADDR + %H40%D;
```

Referencing Data in a Flat Segment

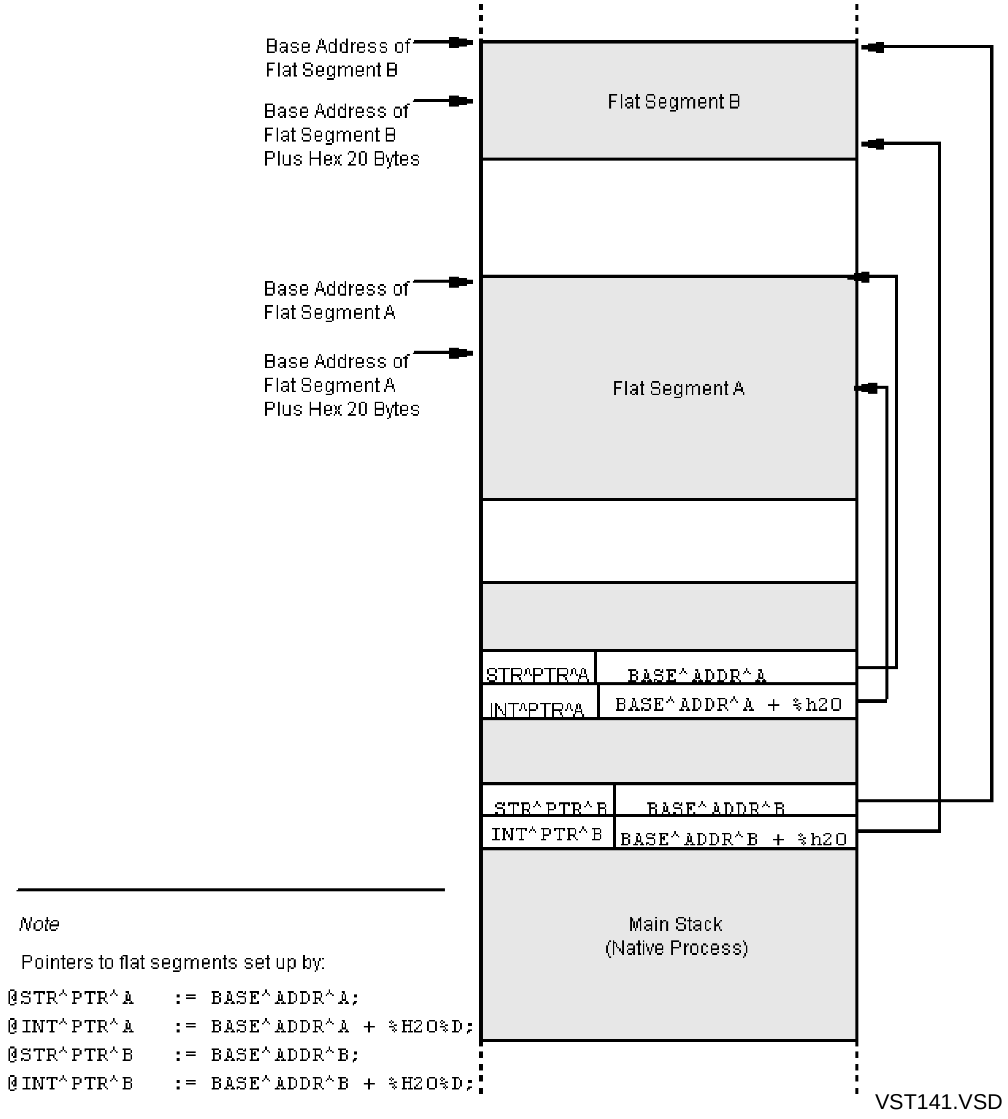
Following are extended pointers for STRING and INT data types. The pointers will be used to access data in two flat segments.

```
INT(32)  BASE^ADDR^A;
STRING  .EXT STR^PTR^A;
INT     .EXT INT^PTR^A;

INT(32)  BASE^ADDR^B
STRING  .EXT STR^PTR^B;
INT     .EXT INT^PTR^B;

@STR^PTR^A   := BASE^ADDR^A ;
@INT^PTR^A   := BASE^ADDR^A  + %H20%D;

@STR^PTR^B   := BASE^ADDR^B ;
@INT^PTR^B   := BASE^ADDR^B  + %H20%D;
```

In the above statements, BASE^ADDR^A and BASE^ADDR^B are the byte addresses of the first locations in the extended data segments and were returned either by the call to SEGMENT_ALLOCATE_ that allocated these segments or by a call to the SEGMENT_GETINFO_ procedure.

Figure 17-8 shows the effect of the pointers declared above.

Figure 17-8. Referencing Flat Segments


Base Address of Flat Segment B
Base Address of Flat Segment B Plus Hex 20 Bytes
Flat Segment B
Base Address of Flat Segment A
Base Address of Flat Segment A Plus Hex 20 Bytes
Flat Segment A
STR^PTR^A
BASE^ADDR^A
INT^PTR^A
BASE^ADDR^A + %h20
STR^PTR^B
BASE^ADDR^B
INT^PTR^B
BASE^ADDR^B + %h20
Main Stack
(Native Process)
VST141.VSD

Note

Pointers to flat segments set up by:

```
@STR^PTR^A    := BASE^ADDR^A;
@INT^PTR^A    := BASE^ADDR^A + %H20%D;
@STR^PTR^B    := BASE^ADDR^B;
@INT^PTR^B    := BASE^ADDR^B + %H20%D;
```

Checking the Size of an Extended Data Segment

To determine the size of a flat segment or a selectable segment (regardless of whether or not the selectable segment is currently in use), supply the SEGMENT_GETINFO_ procedure with the appropriate segment ID.

The following statement returns the size of extended data segment 3 in the variable SEGMENT^SIZE:

```
SEGMENT^ID := 3;
ERROR := SEGMENT_GETINFO_(SEGMENT^ID,SEGMENT^SIZE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

The variable to contain the returned segment size value must be a 32-bit integer. The size of the segment is expressed in bytes.

Changing the Size of an Extended Data Segment

You can alter the size of an extended data segment by calling the RESIZESEGMENT procedure. You supply the procedure with the segment ID and the new segment size. The following example allocates a selectable segment and enlarges it from 8000 bytes to 20000 bytes:

```
SEGMENT^ID := 1;
SIZE := 8000D;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           SIZE,
                           !swap^file:length!,
                           ERROR^DETAIL,
                           !pin!,
                           !segment^type!,
                           BASE^ADDRESS
                           MAX^SIZE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
  .
  .
NEW^SEGMENT^SIZE := 20000D;
ERROR := RESIZESEGMENT(SEGMENT^ID, NEW^SEGMENT^SIZE);
IF ERROR <> 0 THEN ...
```

If a flat segment will be resized, the maximum segment size (1120 megabytes in the TNS/R environment and 1536 megabytes in the TNS/E environment) should be declared in the SEGMENT_ALLOCATE_ procedure call that allocates the segment. This prevents other flat segments from allocating the same space before the resizing is performed. To reserve the maximum segment size, use the *`max-size`* parameter in the SEGMENT_ALLOCATE_ procedure call. The *`max-size`* parameter defines the upper limit of the *`new-segment-size`* parameter of the RESIZESEGMENT procedure. The following example allocates an 8000-byte flat segment and specifies a maximum segment size of 64000 bytes. Later in the program, the segment is resized to its maximum size.

```
SEGMENT^ID := 1;
SIZE := 8000D;
MAX^SIZE := 64000D;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           SIZE,
                           !swap^file:length!,
                           ERROR^DETAIL,
                           !pin!,
                           !segment^type!,
                           BASE^ADDRESS[I],
                           MAX^SIZE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
 .
 .
ERROR := RESIZESEGMENT (SEGMENT^ID, MAX^SIZE);
IF ERROR <> 0 THEN ...
```

Note. RESIZESEGMENT is a resource-intensive procedure. You should therefore avoid frequent calls to this procedure. A general guideline is to call RESIZESEGMENT only when changing the size of a segment by more than 128 K bytes. Resizing a segment by less than 20% should also be avoided.

See the *Guardian Procedure Calls Reference Manual* for complete details of the RESIZESEGMENT procedure.

Transferring Data Between an Extended Data Segment and a File

Transferring data between extended data segments and a file is like transferring data between the TNS user data segment and a file. However, you must use only procedures with the suffix “X” (for example, READX or WRITEX).

To improve the performance of I/O that involves extended data memory, you can use SETMODE function 141 to transfer larger blocks of data at once.

The following paragraphs discuss both the procedures that perform I/O with extended data segments and SETMODE 141. For complete details on these procedures, see the *Guardian Procedure Calls Reference Manual*.

Using I/O Procedure Calls With Extended Data Segments

I/O procedures such as READ, READUPDATE, WRITE, and so on cannot access data outside the TNS user data segment or, in native processes, the globals-heap and stack segments. To transfer data between extended data segments and disk files or processes, you must use the I/O procedures whose names end with the letter “X.”

The I/O procedures that can access extended data segments are READX, READUPDATEX, REPLYX, WRITEX, WRITEREADX, WRITEUPDATEX, READLOCKX, WRITEUPDATEUNLOCKX, and AWAITIOX. These procedures are similar in function to their counterparts for nonextended data segments; however, these procedures allow the I/O buffer to be in either an extended data segment or the TNS user data segment.

For example, the following statements transfer 10 bytes of data from the beginning of the extended data segment to the terminal:

```
INT(32) .EXT EXT^PTR := BASE^ADDRESS;
 .
 .
WCOUNT := 10;
CALL WRITEX(TERM^NUM,EXT^PTR,WCOUNT);
```

In the above example, BASE^ADDRESS is the byte address of the beginning of the extended data segment. This value was returned either by the call to SEGMENT_ALLOCATE_ that allocated this segment or by a call to the SEGMENT_GETINFO_ or SEGMENT_USE_ procedure.

Transferring Large Buffers Quickly

It is usually safer to handle large buffers in an extended data segment (or in the upper half of the TNS user data segment) than it would be to buffer such data in the data stack itself; you avoid running the risk of filling up the data stack with your large buffer. To help perform data transfers to such buffers efficiently, you can use SETMODE function 141.

SETMODE function 141 enables and disables transfers of larger blocks of data between extended data segments and a disk file. Using SETMODE function 141 with other I/O procedures, you can transfer up to 56K bytes of data between extended data segments and a DP2 disk file that has been opened for unstructured access. (See Section 5, Communicating With Disk Files, for a discussion of unstructured file access.) The amount of data transferred must be a multiple of 2048 bytes.

Note. Some Expand connections only support up to 30K bytes of data for a transfer between extended data segments and a DP2 disk file that has been opened for unstructured access.

To enable large transfers, call the SETMODE procedure as follows:

```
LITERAL LARGE^TRANSFERS = 141,
        ENABLE          =   1,
        DISABLE         =   0;
 .
 .
CALL SETMODE(FILE^NUMBER,
             LARGE^TRANSFERS,
             ENABLE);
```

After the call to SETMODE function 141, the only I/O procedure calls permitted against the file are calls to READX, READUPDATEX, WRITEX, and WRITEUPDATEX. These procedures can then transfer up to 56K bytes at once.

Transferring Large Buffers Using Nowait I/O

You can perform nowait I/O when large buffers are enabled. However, your program must not refer to data in the I/O buffer until the I/O operation is complete.

To check for completion of a nowait operation on an extended data segment, use the AWAITIOX procedure.

Moving Data Between Extended Data Segments

The method to use for transferring data between two extended data segments depends on whether the segments are selectable segments or flat segments.

To move data between flat segments, or between a flat segment and the current selectable segment, no special procedure call is needed; you can use normal assignment or move statements. To move data between segments where at least one of the segments is a not-current selectable segment, use the MOVEX procedure. (When using the MOVEX procedure, it does not matter whether either the source or target selectable segment is currently in use.)

Note, however, that MOVEX is a time-consuming procedure, and you should therefore avoid using it wherever possible.

When calling the MOVEX procedure, you must specify where you want to move data from, where you want to move data to, and how much data you want to move. You specify the source and target addresses of the move by indicating the segment ID and the starting address. You specify the amount of data you want to move as a number of bytes.

The following example allocates four selectable segments and moves 512 bytes from the beginning of segment 0 to the beginning of segment 1:

```
FOR I := 0 TO 3 DO
BEGIN
   SEGID[I] := I;
   SIZE := 100000D;
   ERROR := SEGMENT_ALLOCATE_(SEGID[I],
                              SIZE,
                              !swap^file:length!,
                              ERROR^DETAIL,
                              !pin!,
                              !segment^type!,
                              BASE^ADDRESS[I]);
   IF ERROR <> 0 THEN CALL ERROR^HANDLER;
END;
 .
 .
BYTE^COUNT        :=512D;
SOURCE^SEGMENT^ID := 0;
TARGET^SEGMENT^ID := 1;
@SOURCE^PTR := BASE^ADDRESS[0];
@TARGET^PTR := BASE^ADDRESS[1];
ERROR := MOVEX(SOURCE^SEGMENT^ID,
               SOURCE^PTR,
               TARGET^SEGMENT^ID,
               TARGET^PTR,
               BYTE^COUNT);
```

If the MOVEX call was successful, then the call returns an error value of 0. Any other value indicates an error. Typical causes of error are a nonexistent data segment (error 2) or an out-of-bounds address (error 22).

The following example uses assignment statements to perform a similar move between flat segments:

```
INT(32) BASE^ADDR1;
INT(32) BASE^ADDR2;
INT(32) .EXT SOURCE^PTR;
INT(32) .EXT TARGET^PTR;
OPTIONS.<14> := 1; !SET OPTION BIT FOR FLAT SEGMENT!
FOR I := 0 TO 9 DO
BEGIN
   SEGID[I] := I;
   SIZE := 8000D;
   ERROR := SEGMENT_ALLOCATE_(SEGID[I],
                              SIZE,
                              !swap^file:length!,
                              ERROR^DETAIL,
                              !pin!,
                              !segment^type!,
                              BASE^ADDRESS[I],
                              !max^size!,
                              OPTIONS);
   IF ERROR <> 0 THEN CALL ERROR^HANDLER;
END;
 .
 .
@SOURCE^PTR := BASE^ADDR1;
@TARGET^PTR := BASE^ADDR2;
TARGET^PTR ":=" SOURCE^PTR FOR 512 BYTES;
```

Checking Address Limits of an Extended Data Segment

You can use the ADDRESS_DELIMIT_ procedure to obtain the addresses of the first and last bytes of a particular area of your logical address space such as a flat segment or a current selectable segment.

You supply an address contained within the address area of interest, passing it to ADDRESS_DELIMIT_ in the value parameter *`address`*. You can use the *`segment-id`* output parameter to obtain the segment ID of the area if it is an extended data segment. You can also use the *`address-descriptor`* output parameter to obtain a set of flags that describe the area.

In the following example, an address contained in an extended data segment is passed to ADDRESS_DELIMIT_. The procedure returns the address of the last byte of the segment and also the segment ID.

```
INT     .EXT MY^EXT^DATA[0:99];
INT(32)  HIGH^ADDR;
INT      ERROR,
         ERROR^DETAIL,
         SEGMENT^ID;
```

```
          .
          .
ERROR := ADDRESS_DELIMIT_ (@MY^EXT^DATA,
                           ! low^address !,
                           HIGH^ADDR,
                           ! address^descriptor ! ,
                           SEGMENT^ID,
                           ERROR^DETAIL);

IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Sharing an Extended Data Segment

Processes that share data can choose to share extended data segments. You do this by setting appropriate parameters in the SEGMENT_ALLOCATE_ procedure call.

Note. Processes that share extended data segments must be in the same CPU, unless the segments are read-only, in which case they can be shared across CPUs.

There are two ways an application process can share segments:

- Using the PIN method
- Using the file-name method

The method you choose will depend on the information your process knows about the process that originally allocated the data segment:

- If your process knows the PIN of the process that allocated the extended data segment and the segment ID that was allocated, then your process can use the PIN method.
- If your process knows the swap-file name that the other process assigned to the extended data segment, then your process can use the file-name method.

The following paragraphs describe each method.

Using the PIN Method

Your process can use the PIN method to share an extended data segment with another process if all the following are true:

- Your process is in the same CPU as the process that allocated the extended data segment.
- Your process knows the PIN of the process that allocated the extended data segment.
- Your process knows the segment ID of the extended data segment.
- Your process has any of the following:
    - The same process access ID as the process that allocated the data segment

    - The process access ID of the group manager for the process access ID of the process that allocated the extended data segment
    - The super ID

To specify sharing using the PIN method, your process must call the SEGMENT_ALLOCATE_ procedure and specify the PIN of the process that allocated the extended data segment, along with the segment ID known by the process that allocated the extended data segment. The process that allocated the segment can determine its own PIN and pass it to the process that calls SEGMENT_ALLOCATE_.

The following example specifies sharing of segment 3 using the PIN method:

```
SEGMENT^ID := 3;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           !segment^size!,
                           !swap^file:name!,
                           !error^detail!,
                           PIN);
```

Note that the segment size must not be specified when sharing by the PIN method.

Using the File-Name Method

Your process can use the file-name method to share the extended data segment of another process if all the following are true:

- Your process is in the same CPU as the process that allocated the extended data segment.
- Your process knows the swap-file name that the process that allocated the extended data segment assigned to it. (The Kernel-Managed Swap Facility swap file, the default, cannot be used for this purpose.)
- Your process has appropriate Guardian security to access the file. See Managing Swap Space earlier in this section.

To specify sharing using the file-name method, your process must call the SEGMENT_ALLOCATE_ procedure and specify the swap-file name of the extended data segment. You must also set the *segment-type* parameter to specify sharing by file name. You can do this using the ZSYS^VAL^SEGALLOCTYPE^DEFFNAME literal from the ZSYSTAL file**.**

The following example specifies segment sharing using the file-name method:

```
SWAP^FILENAME ':=' "$PROGRAM.SWPFILES.MYPROG" -> @S^PTR;
SWAP^FILENAME^LEN := @S^PTR '-' @SWAP^FILENAME;
SEGMENT^TYPE := ZSYS^VAL^SEGALLOCTYPE^DEFFNAME;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           !segment^size!,
                           SWAP^FILENAME:SWAP^FILENAME^LEN,
                           !error^detail!,
                           !pin!,
                           SEGMENT^TYPE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Note that it is not necessary to specify the segment size, because the segment already exists.

Considerations for Sharing a Flat Segment

Flat segments can be shared only with flat segments allocated with specific segment IDs.

If you do not specify a base address, by default the system attempts to map the shared segment starting at the base address specified in the original SEGMENT_ALLOCATE_call (if that process still has the segment allocated, if not then the address is selected from one of the other processes that shares the segment). If the range of the requested segment is partially or completely overlapped in the current process, then an error is returned. If option bit <9> is set to one and the base address is not specified, then the system behaves the same as the default. However, instead of returning an error if the range of the requested segment is partially or completely overlapped in the current process, the system will attempt to allocate the segment at any address within the flat segment space.

Considerations for Sharing a Selectable Segment

Selectable segments can be shared only with selectable segments.

HP recommends against using explicit selectable segments in TNS processes that use the automatic compiler-generated selectable segment, especially in C, Fortran or Pascal programs that use the (default) XMEM memory model. See Using Selectable Segments in TNS Processes.

Determining the Starting Address of a Flat Segment

In certain situations, you may want to find out the starting address of a previously allocated flat segment. For example, you may have a library routine that needs the starting address of a flat segment that was allocated in a previous invocation of the routine.

The starting address of a flat segment is returned by the SEGMENT_ALLOCATE_ procedure call that allocates the segment. To find out the starting address later in a program, use the SEGMENT_GETINFO_ procedure. The following example returns the base address of flat segment 3:

```
INT(32) .EXT SEG^PTR;
.
.
.
SEGMENT^ID := 3;
ERROR := SEGMENT_GETINFO_(SEGMENT^ID,
                          !segment^size!,
                          !filename:maxlen!,
                                 !length!,
                                 ERROR^DETAIL,
                                 @SEG^PTR);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Deallocating an Extended Data Segment

When you have finished accessing an extended data segment, you can deallocate it by supplying the appropriate segment ID to the SEGMENT_DEALLOCATE_ procedure. The SEGMENT_DEALLOCATE_ procedure returns an error value of 0 if the deallocation was successful; any other value indicates that the operation was unsuccessful.

The following example deallocates extended data segments:

```
FOR I := 0 TO 9 DO
BEGIN
   ERROR := SEGMENT_DEALLOCATE_(I,
                                !flags!,
                                ERROR^DETAIL);
   IF ERROR <> 0 THEN...
END;
```

Once the segment is deallocated, the segment no longer exists in physical memory. The swap file, however, retains the segment data if it is a permanent file. If a temporary file is used as the swap file, or if the Kernel-Managed Swap Facility (KMSF) managed the swap space, then the swap data is discarded.

Using Memory Pools

Memory pools provide a mechanism to help you manage extended data segments. A memory pool is an area of an extended data segment, user data segment (TNS processes), or globals-heap segment (TNS/R native processes) that your process allocates and from which your process can obtain and release blocks of storage.

To use memory pools in extended data segments, you must perform the following steps:

1. Allocate an extended data segment and, if the segment is a selectable segment, make it the current segment. (SEGMENT_ALLOCATE_ and SEGMENT_USE_ procedures). (A flat segment does not need to be made current.)
2. Define all or a contiguous part of that extended data segment to be part of a memory pool (POOL_DEFINE_ procedure).
3. Obtain blocks of storage from the memory pool when needed (POOL_GETSPACE_ procedure or POOL_GETSPACE_PAGE_ procedure) and return those storage blocks to the memory pool when no longer needed (POOL_PUTSPACE_ procedure). You can later extend the pool with the POOL_RESIZE_ procedure.

For information on segment allocation and specifying a current segment, see Read-only segments cannot be extensible. and Making a Selectable Segment Current, earlier in this section. The following paragraphs describe how to define a memory pool and how to obtain storage from a memory pool.

Defining a Memory Pool

Use the POOL_DEFINE_ procedure to define a memory pool. You must supply the POOL_DEFINE_ procedure with the address in the current extended data segment or user data area where the pool is to begin and with the size in bytes of the pool.

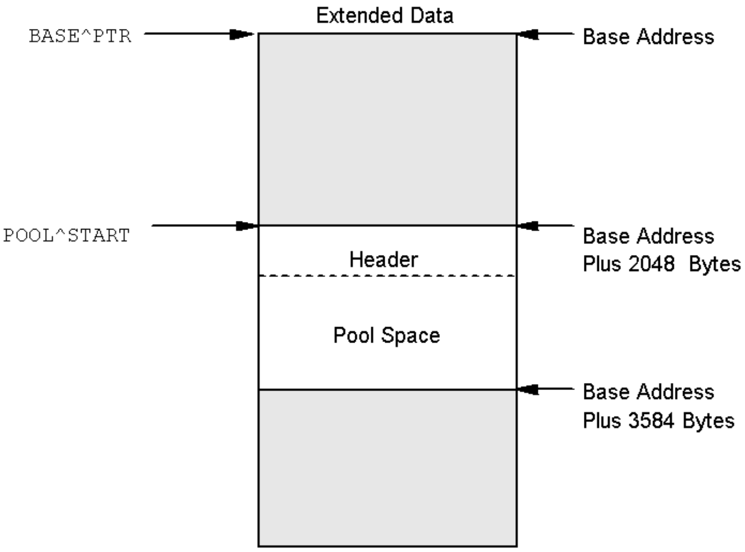
A header is automatically allocated at the beginning of every memory pool. The header, which may vary in length, contains information used by the operating system to manage the pool; it is not intended to be accessed by user programs. In addition, a small portion of the memory pool is used for system overhead. Thus, the total size of the pool available to user programs is somewhat less than the defined size of the pool.

The POOL_DEFINE_ procedure allows you to specify the alignment of the memory pool. Possible alignments are 4-byte, 8-byte, or 16-byte alignment. The default is 16-byte alignment, but you may want to specify a different alignment for efficiency or performance reasons.

The POOL_DEFINE_ procedure returns an error status value that indicates whether the operation was successful. A status value of 0 is returned for a successful operation.

Figure 17-9 shows how a memory pool is allocated from an extended data segment.

Figure 17-9. Defining a Memory Pool


Extended Data
BASE^PTR
Base Address
POOL^START
Header
Base Address
Plus 2048 Bytes
Pool Space
Base Address
Plus 3584 Bytes

Note

```
@POOL^START := @BASE^PTR[1024];
STATUS := POOL_DEFINE_(POOL^START,1536D);
```

VST091.VSD

The following example sets up the memory pool shown in Figure 17-9. The example allocates an extended data segment, makes that segment current, and defines a pool within that segment:

```
INT      ERROR;
INT      OPTIONS;
INT      SEGMENT^ID;
INT      OLD^SEGMENT^ID;
INT(32)  SEGMENT^SIZE;
INT      ERROR^DETAIL;
INT(32)  BASE^ADDRESS;
INT .EXT BASE^PTR := BASE^ADDRESS;
INT .EXT POOL^START;
INT(32)  MAX^POOLSIZE;
 .
 .
!Allocate a flat segment of 4000 bytes.
OPTIONS.<14> := 1;
SEGMENT^ID := 0;
SEGMENT^SIZE := 4000D;
ERROR := SEGMENT_ALLOCATE_(SEGMENT^ID,
                           SEGMENT^SIZE,
                           !swap^file:length!,
                           ERROR^DETAIL,
                           !pin!,
                           !segment^type!,
                           BASE^ADDRESS,
                           !max^size!,
                           OPTIONS);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;

!Make the segment current (if a selectable segment).
SEGMENT^ID := 0;
ERROR := SEGMENT_USE_(SEGMENT^ID,
                      OLD^SEGMENT^ID,
                      @BASE^PTR,
                      ERROR^DETAIL);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;


!Define a 1536-byte memory pool starting 1024 words into the
!extended data segment.
@POOL^START := @BASE^PTR[1024];
MAX^POOLSIZE := 1536D;
ERROR := POOL_DEFINE_(POOL^START,MAX^POOLSIZE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
.
.
```

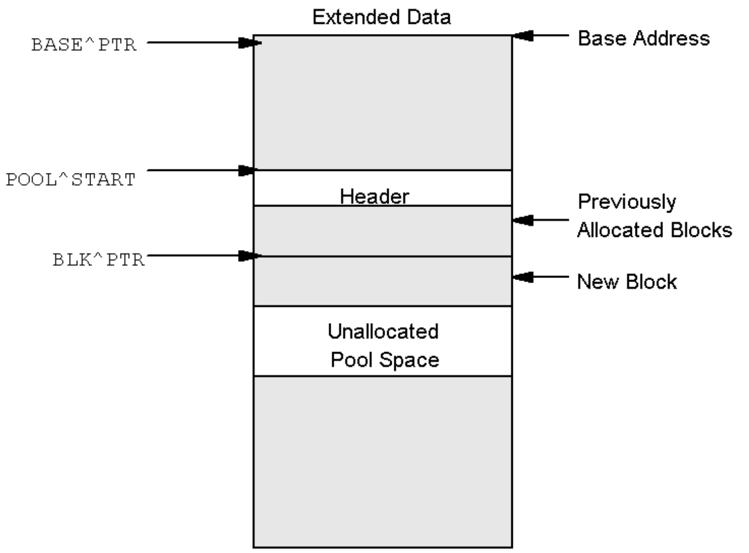
Getting Space in a Memory Pool

After defining a pool, your process can obtain blocks of space from that pool by calling the POOL_GETSPACE_ procedure. You must specify the pool from which you want to obtain blocks by indicating the starting address of the pool. You must also specify the size of the block you require in bytes.

The POOL_GETSPACE_ procedure allocates the first block of contiguously available memory of the size requested that lies within the memory pool. The POOL_GETSPACE_ procedure then returns the address of the block of memory allocated. You can assign this address to an extended pointer so that you can use it to refer to locations in the block. If an error (such as insufficient space left in the memory pool) occurs, then the procedure returns NIL_ instead of an address.

Figure 17-10 shows how a block of storage is obtained from a memory pool.

Figure 17-10. Getting Space in a Memory Pool



Note

`@BLK^PTR := POOL_GETSPACE_(POOL^START,BLK^SIZE);`

VST092.VSD

In the following example, the POOL_GETSPACE_ procedure obtains a 64-byte block of memory from the memory pool defined in the example under Defining a Memory Pool; the address of that block is stored in BLK^PTR (as shown in Figure 17-8), which is then used to refer to locations in the block.

```
  .
  .
!Obtain a 64-byte storage block from the memory pool.
BLK^SIZE := 64D;
```

```
@BLK^PTR := POOL_GETSPACE_ (POOL^START,
                            BLK^SIZE,
                            ERROR);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
  .
  .
BLK^PTR[4] := 12;
```

Returning Memory Pool Space

When your process no longer needs a block of space it obtained from a memory pool, your process can return the block to the memory pool by calling the POOL_PUTSPACE_ procedure. Once a block of data is returned to the memory pool, that storage space becomes available for assignment to other storage blocks.

You must supply the POOL_PUTSPACE_ procedure with the starting address of the pool from which the block of memory was obtained. You must also supply the starting address of the block you are returning.

In the following example, the POOL_PUTSPACE_ procedure returns the block pointed to by BLK^PTR to the memory pool:

```
ERROR := POOL_PUTSPACE_ (POOL^START,BLK^PTR);
```

Changing the Size of a Memory Pool

When you define a memory pool, you specify the size of that memory pool. You can later change the size of a pool by calling the POOL_RESIZE_ procedure.

To change the size of a memory pool, you must supply the POOL_RESIZE_ procedure with the starting address of the pool. You must also specify the new size of the memory pool in bytes.

The POOL_RESIZE_ procedure returns an error status. If the error status is zero, then the operation was successful. If a nonzero value is returned, then the operation failed. Some reasons for failure are that the requested size would shrink the pool so much that allocated storage blocks would no longer remain within the pool (error 4), or that the new size would cause a bounds error, such as expanding the pool beyond the end of the extended data segment (error 3).

The following example changes the size of the pool identified by POOL^START. After the change, the memory pool size is 2048 bytes.

```
INT(32) NEW^POOLSIZE;
INT .EXT POOL^START;
  .
  .
NEW^POOLSIZE := 2048D;
ERROR := POOL_RESIZE_ (POOL^START,NEW^POOLSIZE);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Note that the POOL_RESIZE_ procedure does not move the pool; it only adds neighboring addresses to the pool. The preceding example assumes that the additional space exists in the segment and is not occupied by other data. If the pool is defined at the end of its segment (that is, there is no room to resize the pool), the RESIZESEGMENT procedure can be used to extend the segment before calling POOL_RESIZE_.

Getting Information About a Memory Pool

Use the POOL_GETINFO_ procedure to return information about a memory pool. Information returned includes error information, pool size, and the number of bytes currently allocated.

To get information about a memory pool, you must supply the POOL_GETINFO_ procedure with the starting address of the pool. You must also supply output parameters for the values to be returned.

The following example returns the size of the pool pointed to by POOL^START. (The size returned will be less than the size specified in the POOL_DEFINE_ procedure because of the space occupied by the pool header.)

```
ERROR := POOL_GETINFO_(POOL^START,
                      !error^detail!,
                       POOL^SIZE);
```

See the *Guardian Procedure Calls Reference Manual* for details on other information returned by POOL_GETINFO_.

Debugging a Memory Pool

The POOL_CHECK_ procedure returns information to help diagnose errors that occur when using memory pools. You typically call POOL_CHECK_ after an error has been returned by one of the other pool-management procedures. Specific information returned by POOL_CHECK_ depends on the particular error. Returned information can include:

- Address of the block of space within the pool where the error occurred.
- Address of the valid block closest to the block where the error occurred.
- Size of the last valid block allocated before the error occurred.

See the *Guardian Procedure Calls Reference Manual* for details on the information returned by POOL_CHECK_.

18 Managing Time

Time management involves two related concepts:

- Creating and manipulating timestamps—that is, finding out what the time is
- Performing timing operations—that is, finding out how much time it takes to perform a given task

This section begins by describing how timestamps are generated, what kind of timestamps are available, and the different ways in which the time of day can be represented; see How the System Keeps Time. The subsection Using the Time Management Procedures introduces the system procedures that you can use to manipulate timestamps. The subsection Time and Date Manipulation goes on to describe the procedures that form the programmatic interface to system timekeeping.

Timing information is described in Timing in Elapsed Time and Timing in Process Time. This subsection explains the difference between elapsed time and process time and describes the tasks you can perform with the related procedures.

The last subsection, Managing System Time, describes the procedures you can use to manipulate system timekeeping. Specifically, this includes getting and setting the system time.

How the System Keeps Time

The basis of all timing performed on the system is a hardware clock that increments every clock cycle. This clock represents the number of microseconds since the last time the node was cold loaded. All other time values are derived from this clock.

Clock Averaging and System Time

Each CPU in the system has its own CPU clock that keeps the time for that CPU. Typically, each CPU clock in the system runs at a slightly different speed. The system determines system time by taking the average of the various CPU times; it then establishes an adjustment value for each CPU clock. The adjustment value, which is periodically updated, enables each CPU to provide the correct system time when queried. Thus, when you ask for a timestamp, what you get is the CPU time corrected by the adjustment value, In this way, the CPU clocks are effectively synchronized.

While the clock-averaging algorithm keeps CPU clocks synchronized with each other, it does not necessarily keep system time consistent with the real time of day. For this purpose, the SYSTEMCLOCK_SET_ or SETSYSTEMCLOCK procedure provides a means for periodically adjusting the system time, adjusting the system time rate, or setting the system time. Managing System Time at the end of this section provides details.

Synchronizing system time to any standard time requires some program, such as a Network Time Protocol (NTP) client, to set or adjust the time and clock rate to match an external reference clock. Typically a reference clock is associated with an NTP server on the LAN or WAN.

HP NonStop™ Time Synchronization (TimeSync) synchronizes the NonStop system clocks for all current NonStop and Neoview systems. It has the ability to act as an NTP client, an NTP server, or both simultaneously. For more information about TimeSync, see the *Time Synchronization User's Guide*."

Time Zones and Daylight Saving Time

Timestamps generated by the system can be presented in any of the standard time representations listed below:

- GMT or Greenwich mean time. This term is obsolete; its value was a popular basis for calculating time throughout the world, based on the mean solar time for the meridian at Greenwich, England. The standard replacement is UTC, Coordinated Universal Time, based on many atomic clocks throughout the world. To preserve the relationship of UTC to solar time, an occasional leap second is added. The NonStop kernel does not implement leap seconds, so HP NonStop documentation still refers to GMT.
- LST or local standard time. This value represents time in the local time zone, without any adjustment for daylight saving time. LST is GMT plus an offset that depends on the time zone.
- LCT or local civil time. This value represents time in the local time zone, including any adjustment for daylight saving time (DST). LCT is LST plus a DST offset.

128-Bit, 64-Bit, and 48-Bit Timestamps

The operating system provides three kinds of timestamps: 64-bit Julian timestamps and 48-bit timestamps in the TNS and native environments. All H-series and J-series RVUs contain procedure calls using the 128-bit timestamp.128-bit timestamps provide timestamps that can be used as unique identifiers across all CPUs in a node.

All 64-bit Julian timestamp and 48-bit timestamp procedure calls and information can be used on TNS/E systems.

- The TNS/E environment supports a 128-bit unique timestamp that returns a unique value on every call within a single EXPAND network. This 128-bit timestamp (returned by the TS_UNIQUE_CREATE_ procedure is based on the following properties:
    - This timestamp is monotonically increasing when accessed on the same CPU.
    - The 128-bit timestamp is globally unique; it will never be the same as any other timestamp returned by the TS_UNIQUE_CREATE_ procedure in any CPU in the same EXPAND network.
- A 64-bit Julian timestamp (returned by the JULIANTIMESTAMP procedure) is a quantity equal to the number of microseconds since January 1, 4713 B.C.,

12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time (GMT), local standard time (LST), or local civil time (LCT). There is no way to examine a Julian timestamp and determine which time it represents.

Related to the Julian timestamp is a 32-bit Julian day number, giving the number of days since January 1, 4713 B.C.

- 48-bit timestamps (returned by the TIMESTAMP procedure) measure the difference between the current local civil time and midnight at the start of December 31, 1974. These timestamps are measured in units of 0.01 second.

Use 64-bit Julian timestamps, rather than 48-bit timestamps when developing new applications on TNS and TNS/R systems. When developing new applications on TNS/E systems, use 64-bit timestamps unless you need a unique timestamp, in which case you should use 128-bit timestamps. Both 64-bit and 128-bit timestamps have a microsecond resolution..

Note. The RCLK instruction ($READCLOCK in TAL) is another source of 64-bit timestamps. It returns a value representing the local civil time in microseconds since midnight (00:00) on 31 December 1974. Note that this is not a Julian timestamp and therefore it is not transferable across HP systems. Applications should avoid using the RCLK instruction except where necessary.

Using the Time Management Procedures

The system provides several procedures that you can use to manipulate 128-bit timestamps, 64-bit timestamps. or 48-bit timestamps. These procedures can convert timestamps into other representations of date and time and perform further manipulations of these representations.

A 128-bit timestamp can:

- Extract the Julian timestamp from the 128-bit timestamp using the TS_UNIQUE_CONVERT_TO_JULIAN_ procedure. Use the results returned from this procedure to compare timestamps generated in different CPUs or on different systems in the same EXPAND network.
- Compare two Unique timestamps generated in the same EXPAND network using the TS_UNIQUE_COMPARE_ procedure. Comparisons are represented as a relative ordering of when the two timestamps were generated.
- Return a value (in nanoseconds) representing the time since coldload using the TS_NANOSECS_ procedure. HP recommends using this procedure only when you need a very fine granularity of time.

A 64-bit timestamp can be converted to

- An eight-word array containing a Gregorian representation of the date and time; that is, the year, month, day of the month, and the time of day down to the number of microseconds. Time in this representation can be either GMT, LST, or LCT.
- A Julian day number.
- An integer value representing the day of the week.

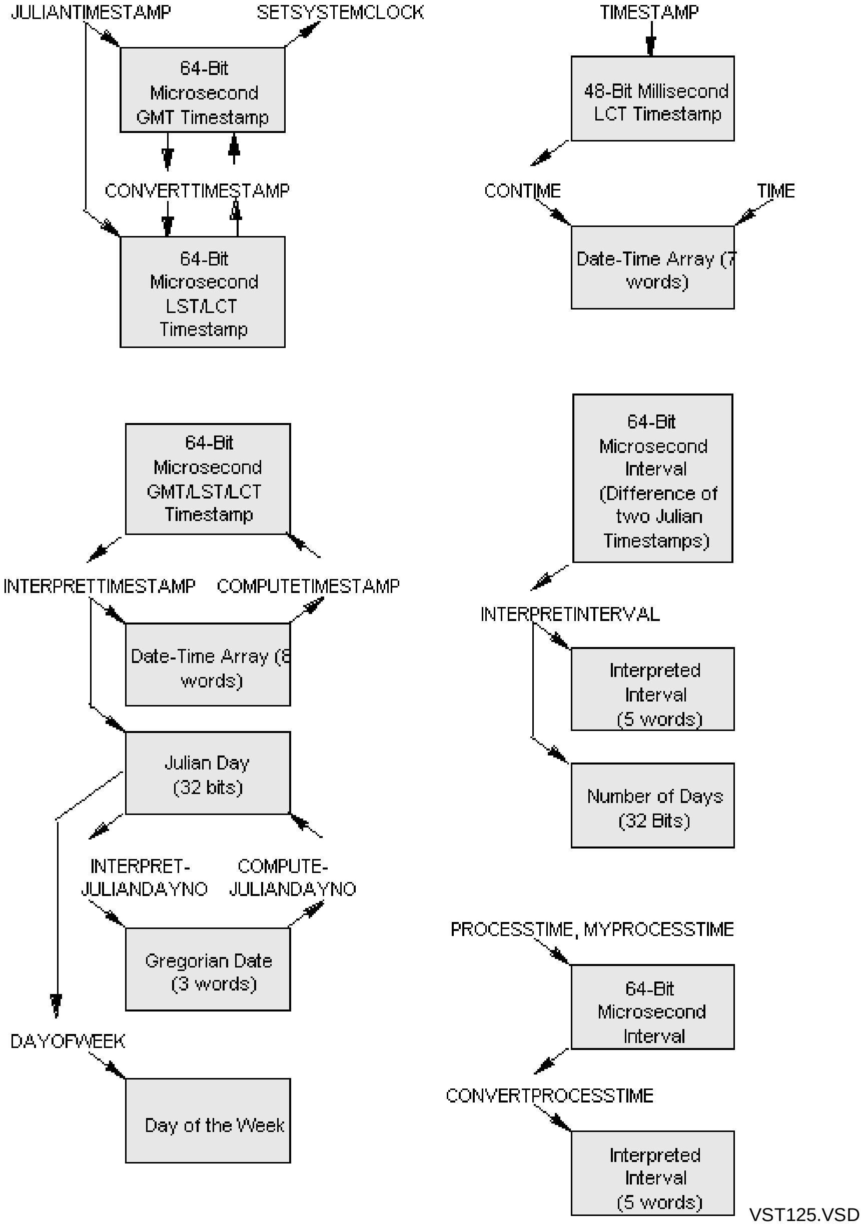
A 48-bit timestamp can be converted to a seven-word array containing a Gregorian representation of the date and time (LCT) in millisecond resolution.

In addition to converting and manipulating timestamps, you can use time management procedures to create and interpret 64-bit intervals. These time intervals measure time in microseconds. Time intervals can simply be a comparison of two 64-bit Julian timestamps, or they can be a measure of CPU time (CPUTIMES procedure), or process time (PROCESSTIME procedure or MYPROCESSTIME procedure).

A time interval can also be represented by five words containing the number of hours, minutes, seconds, milliseconds, and microseconds.

Figure 18-1 shows the relationships between the various representations of time, time intervals, and the system procedures that manipulate them for TNS and TNS/R systems. Use the TS_UNIQUE_CONVERT_ procedure to manipulate a 128-bit timestamps.

Figure 18-1. Time Management Procedures


JULIANTIMESTAMP
SETSYSTEMCLOCK
64-Bit Microsecond GMT Timestamp
CONVERTTIMESTAMP
64-Bit Microsecond LST/LCT Timestamp
TIMESTAMP
48-Bit Millisecond LCT Timestamp
CONTIME
TIME
Date-Time Array (7 words)
64-Bit Microsecond GMT/LST/LCT Timestamp
INTERPRETTIMESTAMP
COMPUTETIMESTAMP
Date-Time Array (8 words)
Julian Day (32 bits)
INTERPRET-JULIANDAYNO
COMPUTE-JULIANDAYNO
Gregorian Date (3 words)
DAYOFWEEK
Day of the Week
64-Bit Microsecond Interval (Difference of two Julian Timestamps)
INTERPRETINTERVAL
Interpreted Interval (5 words)
Number of Days (32 Bits)
PROCESSTIME, MYPROCESSTIME
64-Bit Microsecond Interval
CONVERTPROCESSTIME
Interpreted Interval (5 words)
VST125.VSD

Time and Date Manipulation

This subsection describes how to use the system procedures that obtain or manipulate Julian timestamps (64 bits) or 48-bit timestamps. These operations include:

- Obtaining timestamps
- Computing a time interval
- Converting between timestamps and a Gregorian representation of the date and the time of day

Working With 64-Bit Julian Timestamps

You should use a Julian timestamp whenever you need to measure a time interval or apply a timestamp to an event.

When you measure a time interval, you need to be sure that no clock adjustments are made during the interval. The Julian timestamp is not affected by daylight saving time.

When applying a timestamp to an event (such as updating a record), you need a common basis for all such timestamps. Again you should use the GMT Julian timestamp not only to avoid confusion during daylight saving transition but also to provide a standard that can be used in different time zones. The following tasks involve Julian timestamps:

- Obtain a Julian timestamp from your local node or from a remote node in the network (JULIANTIMESTAMP procedure).
- Time an interval using Julian timestamps (JULIANTIMESTAMP procedure).
- Convert a Julian timestamp into a Gregorian date and time of day (INTERPRETTIMESTAMP procedure).
- Convert a Gregorian date and time of day into a Julian timestamp (COMPUTETIMESTAMP procedure).
- Convert Julian timestamps between local time and GMT (CONVERTTIMESTAMP procedure).
- Convert a period of time specified in microseconds into a number of hours, minutes, seconds, milliseconds, and microseconds (INTERPRETINTERVAL procedure).

The following paragraphs describe how to perform these tasks.

Obtaining a Julian Timestamp: Local Node

To obtain a Julian timestamp, you call the JULIANTIMESTAMP procedure. This procedure can return GMT for the current time, GMT at the last system cold load, GMT at the last system generation, or the number of microseconds since the last cold load. You choose the timestamp you want by setting the *type* parameter as follows:

0	The current GMT
1	GMT at the last system cold load
2	GMT at the last system generation
3	Microseconds since cold load

The current GMT is the default timestamp.

The following example returns the current GMT and the GMT at the last cold load:

```
LITERAL COLD^LOAD       = 1,
        SYSGEN^GMT      = 2,
        SINCE^COLD^LOAD = 3;
FIXED GMT^TIME,
      COLD^LOAD^TIME;
INT   TYPE;
 .
 .
GMT^TIME := JULIANTIMESTAMP;
TYPE := COLD^LOAD;
COLD^LOAD^TIME := JULIANTIMESTAMP(TYPE);
```

Timing an Interval Using Julian Timestamps

To time an interval you should use the Julian timestamp because it is important that daylight saving time is not adjusted during the interval. However, comparing two GMT timestamps does not ensure that no clock adjustments have been made. The system time could have been reset by the SETTIME command or the SYSTEMCLOCK_SET_ or SETSYSTEMCLOCK procedure between the two measurements.

Using the JULIANTIMESTAMP procedure with the type parameter set to 3 returns the number of microseconds since cold load (instead of a GMT Julian timestamp). This value is not affected by setting the system time. Comparing this kind of timestamp at the start of the interval with the same kind of timestamp at the end of the interval always yields the length of the interval in microseconds. For example:

```
LITERAL COLD^LOAD       = 1,
        SYSGEN^GMT      = 2,
        SINCE^COLD^LOAD = 3;
FIXED   TIME1,
        TIME2,
        INTERVAL;
INT     TYPE;
 .
 .
```

```
TYPE := SINCE^COLD^LOAD;
TIME1 := JULIANTIMESTAMP(TYPE);
 .
 .
TYPE := SINCE^COLD^LOAD;
TIME2 := JULIANTIMESTAMP(TYPE);

INTERVAL := TIME2 - TIME1;
```

Obtaining a Julian Timestamp: Remote Node

If you are dealing with timestamps on a remote node, then the relevant Julian timestamp is the one that is generated on that node. This is because system time on the remote node may be different from system time on the local node; the operating system makes no attempt to synchronize clocks between nodes.

If, for example, you want to know how much time has passed since a specific file on a remote node was last updated, you would find out using the last update timestamp on the file and the current timestamp from the remote node.

When establishing the current time on a remote node, you should attempt to compensate for the time it takes to send the message containing the timestamp to the local node. You find this out using the following sequence:

1. Call the JULIANTIMESTAMP procedure for the local node to return the number of microseconds since cold load.
2. Call the JULIANTIMESTAMP procedure for the remote node.
3. Call the JULIANTIMESTAMP procedure for the local node to return the number of microseconds since cold load.
4. Compute the difference between the timestamps returned in Steps 1 and 3. This is the time taken to send a message from the local node to the remote node and then to send a message back to the local node from the remote node.
5. Divide the time delay indicated in Step 4 by 2 to get the time to send a message in one direction.
6. Add the delay indicated in Step 5 to the remote timestamp returned in Step 2. The result is a current timestamp for the remote node.

Note. The above algorithm yields an approximate result. The error in the result can be as large as the value computed in Step 5.

The following example calculates the time since the last update of a file named DFILE on a remote node named \SYS2.

```
LITERAL GET^TIME^OF^LAST^UPDATE = 144,
        CURRENT^GMT             =   0,
        SINCE^COLD^LOAD         =   3;

INT    .RESULT[0:3],
        ITEM^LIST;

INT     LENGTH,
        NUMBER^OF^ITEMS,
        RESULTMAX,
        ERROR,
        NODE^NAME[0:3],
        NODE^NUMBER,REMOTE^ERROR,
       .S^PTR;
FIXED   TIME^OF^LAST^UPDATE = RESULT;
FIXED   TIME^BEFORE,
        REMOTE^TIME,
        TIME^AFTER,
        DELAY^TIME,
        REMOTE^GMT,
        TIME^SINCE^LAST^UPDATE;

STRING  FILENAME[0:ZSYS^VAL^LEN^FILENAME - 1];
 .
 .
!Get time of last update:
FILENAME ':=' "\SYS2.$APPLS.FILES.DFILE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILENAME;
ITEM^LIST := GET^TIME^OF^LAST^UPDATE;
NUMBER^OF^ITEMS := 1;
RESULTMAX := 8;
ERROR := FILE_GETINFOLISTBYNAME_(FILENAME:LENGTH,
                                 ITEMLIST,NUMBER^OF^ITEMS,
                                 RESULT,RESULTMAX);
IF ERROR <> 0 THEN CALL ERROR^HANDLER;

!Get remote node number:
NODE^NAME ':=' "\SYS2   ";
CALL LOCATESYSTEM(NODE^NUMBER,
                  NODE^NAME);

!Get time on local node:
TIME^BEFORE := JULIANTIMESTAMP(SINCE^COLD^LOAD);

!Get time on remote node:
REMOTE^TIME := JULIANTIMESTAMP(CURRENT^GMT,
                               !time^update^id!,
                               REMOTE^ERROR,NODE^NUMBER);
IF REMOTE^ERROR <> 0 THEN CALL ERROR^HANDLER;

!Get time again on local node:
TIME^AFTER := JULIANTIMESTAMP(SINCE^COLD^LOAD);

!Compute remote timestamp:
IF TUID1 = TUID2 THEN
BEGIN
   DELAY^TIME := TIME^AFTER - TIME^BEFORE;
   REMOTE^GMT := REMOTE^TIME + (DELAY^TIME/2F);
```

```
  !Compute time since last update:
  TIME^SINCE^LAST^UPDATE := REMOTE^GMT - TIME^OF^LAST^UPDATE;
END;
```

Converting Between a Julian Timestamp and a Gregorian Date and Time

To obtain a Gregorian date and the time of day from a Julian timestamp, you supply the INTERPRETTIMESTAMP procedure with the Julian timestamp. The procedure returns the Gregorian date and the time of day (in Greenwich mean time) in the *`date-and-time`* parameter, as well as the Julian day number in the returned value. The following statement shows an example:

```
INT DATE^AND^TIME[0:7];
FIXED JULIAN^TIMESTAMP;
INT(32) JULIAN^DAY^NUMBER;
 .
 .
JULIAN^TIMESTAMP := JULIANTIMESTAMP(CURRENT^GMT);
JULIAN^DAY^NUMBER := INTERPRETTIMESTAMP(JULIAN^TIMESTAMP,
                                        DATE^N^TIME);
```

The eight-word *`date-and-time`* parameter contains the information shown below. Values in parentheses indicate the range of valid values:

```
DATE^AND^TIME[0]   !Gregorian year such as 1990      (1-4000)
DATE^AND^TIME[1]   !Gregorian month                  (1-12)
DATE^AND^TIME[2]   !Gregorian day of the month       (1-31)
DATE^AND^TIME[3]   !Hour of the day                  (0-23)
DATE^AND^TIME[4]   !Minute of the hour               (0-59)
DATE^AND^TIME[5]   !Second of the minute             (0-59)
DATE^AND^TIME[6]   !Millisecond of the second        (0-999)
DATE^AND^TIME[7]   !Microsecond of the millisecond   (0-999)
```

To obtain a Julian timestamp from a Gregorian date and the time of day, you supply the COMPUTETIMESTAMP procedure with the eight-word Gregorian date and time of day in the *`date-n-time`* parameter. The following example converts the date April 11, 1990, at 1:43 p.m. into a 64-bit Julian timestamp:

```
DATE^AND^TIME[0] := 1990;   !year
DATE^AND^TIME[1] := 4;      !month
DATE^AND^TIME[2] := 11;     !day
DATE^AND^TIME[3] := 13;     !hour
DATE^AND^TIME[4] := 43;     !minute
DATE^AND^TIME[5] := 0;      !second
DATE^AND^TIME[6] := 0;      !millisecond
DATE^AND^TIME[7] := 0;      !microsecond
JULIAN^TIMESTAMP := COMPUTETIMESTAMP(DATE^N^TIME,
                                     ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL INVALID^DATE;
```

The above example uses the *`errormask`* parameter to check the validity of the input. If any part of the Gregorian date or time is outside the valid range, then the corresponding bit is set in the *`errormask`* parameter. For example, if the year is outside the range 1 through 4000, then bit 0 (the most significant bit) is set to 1; if the month is specified outside the range 1 through 12, then bit 1 is set; and so on.

Converting a GMT Timestamp Into Local Time

To convert a Julian timestamp representing GMT into a Julian timestamp representing local time, or to convert a local Julian timestamp into a GMT Julian timestamp, you can use the CONVERTTIMESTAMP procedure. The local time used by this procedure can be local standard time (no adjustment made for daylight saving time) or local civil time (time adjusted for daylight saving time). Moreover, the CONVERTTIMESTAMP procedure can work with local time on any network node.

The following example converts GMT into LCT for the local node:

```
LITERAL GMT^TO^LCT = 0,
        GMT^TO^LST = 1,
        LCT^TO^GMT = 2,
        LST^TO^GMT = 3;
INT     NODE^NUMBER,
        NODE^NAME[0:3];

LOCAL^CIVIL^TIME := CONVERTTIMESTAMP(JULIAN^GMT^TIMESTAMP,
                                     GMT^TO^LCT);
```

The next example converts the LCT on the network node named \SYS3 into GMT:

```
NODE^NAME ':=' "\SYS3   ";
CALL LOCATESYSTEM(NODE^NUMBER,NODE^NAME);
GREENWICH^MEAN^TIME := CONVERTTIMESTAMP(LOCAL^CIVIL^TIME,
                                        LCT^TO^GMT,
                                        NODE^NUMBER);
```

Converting Microseconds Into Days, Hours, Minutes, Seconds, Milliseconds, and Microseconds

You can convert a time period into a number of days, hours, minutes, seconds, milliseconds, and microseconds using the INTERPRETINTERVAL procedure. For example, you can compute the difference between two Julian timestamps and then convert the result into a more readable form as follows:

```
LITERAL SINCE^COLD^LOAD = 3;

INT(32) DAYS;
INT     HOURS,
        MINUTES,
        SECONDS,
        MILLISECS,
        MICROSECS;
FIXED   TIME1,
        TIME2,
        INTERVAL;
  .
  .
```

```
TIME1 := JULIANTIMESTAMP(SINCE^COLD^LOAD);
 .
 .
TIME2 := JULIANTIMESTAMP(SINCE^COLD^LOAD);

INTERVAL := TIME2 - TIME1;

DAYS := INTERPRETINTERVAL(INTERVAL,HOURS,
                          MINUTES,SECONDS,
                          MILLISECS,MICROSECS);
```

Working With Julian Day Numbers

For operations that require the date but not necessarily the time of day, you can measure time using Julian day numbers. A Julian day number is the number of days since January 1, 4713 B.C.

You can use system procedures to perform the following operations on Julian day numbers:

- Obtain the Julian day number from a Julian timestamp (INTERPRETTIMESTAMP procedure)
- Convert Julian day numbers into Gregorian dates (INTERPRETJULIANDAYNO procedure)
- Compute the Julian day number from a Gregorian date (COMPUTEJULIANDAYNO procedure)
- Determine the day of the week that corresponds to a Julian day number (DAYOFWEEK procedure)

The following paragraphs describe how to perform these operations.

Obtaining the Julian Day Number

You use the INTERPRETTIMESTAMP procedure to establish the Julian day number. You have already seen how this procedure converts a Julian timestamp into a Gregorian date and time. Here, however, you will examine the return value that contains the Julian timestamp. For example:

```
JULIAN^TIMESTAMP := JULIANTIMESTAMP(CURRENT^GMT);
JULIAN^DAY^NUMBER := INTERPRETTIMESTAMP(JULIAN^TIMESTAMP,
                                        DATE^AND^TIME);
```

Converting Between Julian Day Numbers and Gregorian Dates

To convert a Julian day number into a Gregorian date, you supply the INTERPRETJULIANDAYNO procedure with the Julian day number. The procedure returns the Gregorian date in the parameters `year`, `month`, and `day`. The Julian day number must be greater than or equal to 1,721,119, and no greater than 3,182,395, which refers to December 31, year 4000 of the Gregorian calendar. The following example returns the current Gregorian date:

```
JULIAN^TIMESTAMP := JULIANTIMESTAMP(CURRENT^GMT);
JULIAN^DAY^NUMBER := INTERPRETTIMESTAMP(JULIAN^TIMESTAMP,
                                        DATE^AND^TIME);

CALL INTERPRETJULIANDAYNO(JULIAN^DAY^NUMBER,
                          YEAR,MONTH,DAY);
```

To convert a Gregorian date into a Julian day number, you supply the COMPUTEJULIANDAYNO procedure with the Gregorian year, month, and day of the month. The procedure returns the Julian day number. For example:

```
YEAR := 1952;
MONTH := 11;
DATE := 9;
JULIANDAYNO := COMPUTEJULIANDAYNO(YEAR,MONTH,DAY,ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL BAD^DATE;
```

The `errormask` parameter provides the result of validity checking of the Gregorian date. Bit 0 (the most significant bit) of the `errormask` parameter is set to 1 if the year is outside the range 1 through 4000; bit 1 is set to 1 if the month is outside the range 1 through 12; and bit 2 is set if the day of the month is outside the range 1 through 31 for a month that has 31 days or outside the range 1 through 30 for a month that has 30 days. For the month of February, bit 2 is set if the date is outside the range 1 through 28 if it is not a leap year, or 1 through 29 if it is a leap year.

Converting a Julian Day Number Into a Day of the Week

You can find out the day of the week of a specified Julian day number using the DAYOFWEEK procedure. You need to supply the DAYOFWEEK procedure with the Julian day number; the procedure returns the day of the week represented by an integer value: 0 represents Sunday, 1 represents Monday, and so on. For example:

```
DAY := DAYOFWEEK(JULIAN^DAY^NUMBER);
```

Working With 48-Bit Timestamps

A 48-bit timestamp measures the time since the start of December 31, 1974. All dates and times are in local civil time, and the unit of measurement is 0.01 second.

You can use a 48-bit timestamp when you are only concerned with LCT. Such a timestamp should not be compared with other timestamps nor referred to from a network node in another time zone. Such a timestamp could be used for displaying the local time.

When working with 48-bit timestamps, you can perform the following operations:

- Obtain a 48-bit timestamp (TIMESTAMP procedure)
- Convert a 48-bit timestamp into a Gregorian date and time (CONTIME procedure)

The following paragraphs describe how to perform these operations.

Obtaining a 48-Bit Timestamp

You obtain a 48-bit timestamp using the TIMESTAMP procedure. This procedure returns the time in 0.01-second units since 00:00 on December 31, 1974, in a three-word array in the *`interval-clock`* parameter:

```
INT INTERVAL^CLOCK[0:2];
 .
 .
CALL TIMESTAMP(INTERVAL^CLOCK);
```

Converting a 48-Bit Timestamp Into a Gregorian Date and the Time of Day

You can convert a 48-bit timestamp into a 7-word Gregorian date and the time of day using the CONTIME procedure. The 7-word array that contains the date and time has the following format:

```
DATE^AND^TIME[0]   Gregorian year (for example, 1990)
DATE^AND^TIME[1]   Month of the year (1 to 12)
DATE^AND^TIME[2]   Day of the month (1 to 31)
DATE^AND^TIME[3]   Hour of the day (0 to 23)
DATE^AND^TIME[4]   Minute of the hour (0 to 59)
DATE^AND^TIME[5]   Second of the minute (0 to 59)
DATE^AND^TIME[6]   Hundredth of the second (0 to 99)
```

The following example converts a 48-bit timestamp generated by the TIMESTAMP procedure into the integer form of the Gregorian date and time of day:

```
INT INTERVAL^CLOCK[0:2],
    DATE^AND^TIME[0:6];
 .
 .
CALL TIMESTAMP(INTERVAL^CLOCK);

CALL CONTIME(DATE^AND^TIME,
             INTERVAL^CLOCK[0],
             INTERVAL^CLOCK[1],
             INTERVAL^CLOCK[2]);
```

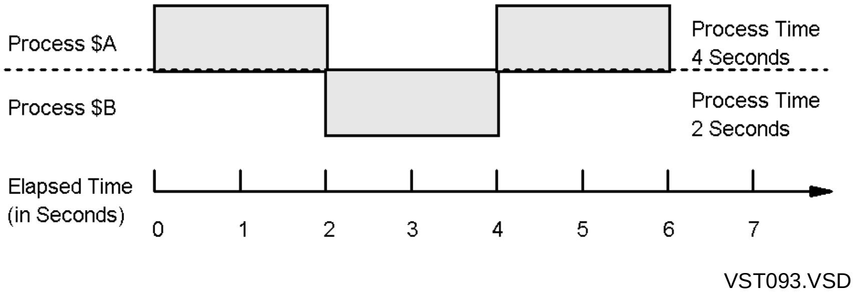
Timing in Elapsed Time and Timing in Process Time

This subsection describes how to start and cancel process timers and perform other timing functions. You can time processes in elapsed time or in process time. Elapsed time is time as measured by the CPU clock, independent of the state of any process.

Process time is the time that a process is active. That is, process time includes the time that the process is executing, whether in user code, system code, or library code. Process time does not include time spent by other processes acting on behalf of your process, nor does it include time spent suspended or waiting for external events. Software interrupts also stop the process timer, but microcode interrupts do not.

Because many processes must share the same CPU, process time and elapsed time will usually be different. Figure 18-2 shows the difference between elapsed time and process time.

Figure 18-2. Elapsed Time and Process Time



You can use time-related operations for setting and canceling timers or for reporting statistical data. A program that sets a timer, however, must also check for timeout; you do this by reading a system message from the $RECEIVE file.

In typical use, you can set a timer to expire if a specific operation does not complete within a period of elapsed time or a period of process time. You can cancel the timer if the operation finishes in time.

An example of a timing function that gathers statistical data is finding out the total process time for a given process.

You can use system procedure calls to perform the following time-out operations on processes. The procedures used here work in 0.01-second units, with an accuracy of plus or minus 0.01 of a second:

- Set a timer to time out after a specific length of elapsed time (SIGNALTIMEOUT procedure in the TNS and native environments. Use the TIMER_START_ procedure in the TNS/E environment for time granularity reflected in microseconds instead of centiseconds.)

- Cancel a timer that runs in elapsed time (CANCELTIMEOUT procedure in the TNS and native environments. Use the TIMER_STOP_ procedure in the TNS/E environment for time granularity reflected in microseconds instead of centiseconds.)
- Set a timer to time out after a specific length of process time (SIGNALPROCESSTIMEOUT procedure)
- Cancel a timer that runs in process time (CANCELPROCESSTIMEOUT procedure)

In addition to the procedures indicated above, several other system procedures, such as AWAITIO and DELAY, take timeout values as parameters. These procedures are supported internally by the timing mechanism discussed above.

The following group of operations reports total process time. These procedures work in units of 1 microsecond (.000001 of a second).

- Establish how much process time your process has used (MYPROCESSTIME procedure)
- Establish how much process time any process has used (PROCESS_GETINFO_ procedure)
- Convert a number of microseconds of process time into a number of hours, minutes, seconds, milliseconds, and microseconds (CONVERTPROCESSTIME procedure)

The following paragraphs describe how to perform these operations.

Setting and Canceling Timers: Elapsed Time

You can set a timer to time out after a specified period of elapsed time using the SIGNALTIMEOUT procedure in the TNS and native environments and the TIMER_START_ procedure in the TNS/E environment. Using SIGNALTIMEOUT, your process will receive system message -22 (the Time signal message) when the timer expires.

You can use the CANCELTIMEOUT or TIMER_STOP_ procedures to cancel a timer started by the SIGNALTIMEOUT or TIMER_START_ procedures. For example, to check that an operation is completed within a certain time, you could start the timer and then start the operation. If the operation finishes within the desired time, you no longer have a need for the timer; you therefore cancel the timer.

To start the timer, supply the SIGNALTIMEOUT procedure with the time period in 0.01-second units. Supply the TIMER_START_ procedure with the time period in 0.000001 second units. You can also supply these procedures with values that will allow the timer to be identified in the message read from $RECEIVE. The SIGNALTIMEOUT or TIMER_START_ procedures return a value in the `tag` parameter for passing to the

CANCELTIMEOUT or TIMER_STOP_ procedures. CANCELTIMEOUT or TIMER_STOP_ uses this value to distinguish between multiple timers.

The Time signal message received from $RECEIVE when the timer expires has the following format:

Format of system message -22 (Time signal message):

```
sysmsg[0]       = -22
sysmsg[1]       = First parameter supplied by SIGNALTIMEOUT;
                  default value 0
sysmsg[2] FOR 2 = Second parameter supplied by SIGNALTIMEOUT;
                  default value 0D
```

Note. There are special considerations when using timers to measure long intervals of elapsed time, such as several hours or more. For information on this topic, refer to the subsection Measuring Long Time Intervals.

The following example starts a timer to expire in one minute. The example uses the `parameter-1` parameter to supply an identifier that will be returned in word 1 of the system message. The SIGNALTIMEOUT procedure returns a value in the `tag` parameter for passing to the CANCELTIMEOUT procedure, which will use it to identify this timer.

```
TIMEOUT^VALUE := 6000D;
PARAMETER^1 := 1;
CALL SIGNALTIMEOUT(TIMEOUT^VALUE,PARAMETER^1,
                   !parameter^2!,
                   TAG^1);
```

Note that `parameter-2` is not supplied in this case. The purpose of `parameter-2` is the same as `parameter-1`, but `parameter-2` allows you to use a 32-bit value instead of a 16-bit value. If used, the value of `parameter-2` is returned in word 2 of the Time signal message.

To cancel the timer set above, supply the CANCELTIMEOUT procedure with the `tag` value that was returned by the SIGNALTIMEOUT procedure:

```
CALL CANCELTIMEOUT(TAG^1);
```

For details on how to read system messages from the $RECEIVE file, see Section 6, Communicating With Processes. You can identify the timer by checking word 1 of the Time signal message; in this case, its value will be equal to `parameter-1`.

Setting and Canceling Timers: Process Time

Setting and canceling timers of process time is like setting and canceling timers of elapsed time, except that the time period measured is limited to the number of milliseconds that the process is active.

You start a process timer using the SIGNALPROCESSTIMEOUT procedure and cancel the timer using the CANCELPROCESSTIMEOUT procedure. Your process will receive system message -26 (the Process time signal message) in its $RECEIVE file when the timer expires.

To start the process timer, supply the SIGNALPROCESSTIMEOUT procedure with the time period in 0.01-second units. You can also supply this procedure with values that will allow the timer to be identified in the message read from $RECEIVE. The SIGNALPROCESSTIMEOUT procedure returns a value in the *tag* parameter for passing to the CANCELPROCESSTIMEOUT procedure. CANCELPROCESSTIMEOUT uses this value to distinguish among multiple timers.

The Process time signal message received from $RECEIVE when the timer expires has the following format:

Format of system message -26 (Process time signal message):

```
sysmsg[0]       = -26
sysmsg[1]       = First parameter supplied by
                  SIGNALPROCESSTIMEOUT;
                  default value 0
sysmsg[2] FOR 2 = Second parameter supplied by
                  SIGNALPROCESSTIMEOUT;
                  default value 0D
```

The following example starts a process timer to expire after 30 seconds of active processing time. The example uses the *parameter-1* parameter to supply an identifier that will be returned in word 1 of the system message. The SIGNALPROCESSTIMEOUT procedure returns a value in the *tag* parameter for passing to the CANCELPROCESSTIMEOUT procedure, which will use it to identify this timer.

```
TIMEOUT^VALUE := 3000D;
PARAMETER^1 := 2;
CALL SIGNALPROCESSTIMEOUT(TIMEOUT^VALUE,
                          PARAMETER^1,
                          !parameter^2!,
                          TAG^1);
```

Note that *parameter-2* is not supplied in this case. The purpose of *parameter-2* is the same as *parameter-1*, but *parameter-2* allows you to use a 32-bit tag instead of a 16-bit tag. If used, the value of *parameter-2* is returned in word 2 of the system message.

To cancel the timer set above, supply the CANCELPROCESSTIMEOUT procedure with the *tag* value that was returned by the SIGNALPROCESSTIMEOUT procedure:

```
CALL CANCELPROCESSTIMEOUT(TAG^1);
```

For details on how to read system messages from the $RECEIVE file, see Section 6, Communicating With Processes. You can identify the timer by checking word 1 of the Process time signal message; in this case, its value will be equal to PARAMETER^1.

Timing Your Process

To find out how much processing time your process has used, you can call the MYPROCESSTIME procedure. This procedure returns the number of microseconds that the process has been active. For example:

```
FIXED PROCESS^TIME;
  .
  .
PROCESS^TIME := MYPROCESSTIME;
```

Timing Another Process

To find out how much processing time has been used by a process other than your own, you can use the PROCESS_GETINFO_ procedure and supply the *`process-time`* parameter. This procedure returns the time in microseconds. For example:

```
FIXED PROCESS^TIME;
  .
  .
ERROR := PROCESS_GETINFO_(PROCESS^HANDLE,
                          !file^name:maxlen!,
                          !file^name^len!,
                          !priority!,
                          !moms^processhandle!,
                          !hometerm:maxlen!,
                          !hometerm^len!,
                          PROCESS^TIME);
```

The procedure returns information about the process identified by the *`process-handle`* parameter. If the process handle is zero or omitted, then the procedure returns information about the current process.

For details about process handles, refer to Section 16, Creating and Managing Processes.

Converting Process Time Into a Readable Form

You can use the CONVERTPROCESSTIME procedure to convert a 64-bit process-time period returned from a call to MYPROCESSTIME or PROCESS_GETINFO_ into a number of hours, minutes, seconds, milliseconds, and microseconds.

The following example converts a 64-bit representation of a period of time:

```
FIXED PROCESS^TIME;
  .
  .
PROCESS^TIME := MYPROCESSTIME;

CALL CONVERTPROCESSTIME(PROCESS^TIME,HOURS,MINUTES,SECONDS,
                        MILLISECONDS,MICROSECONDS);
```

Measuring Long Time Intervals

The SIGNALTIMEOUT procedure measures elapsed time according to the internal clock of the CPU in which the calling process is executing. Typically, the CPU clocks in a system run at slightly different speeds. Recall that the system determines system time by taking the average of all the CPU times in the system, and then establishes adjustment values for the various CPU clocks in order to synchronize them.

Elapsed time, which is measured by a CPU clock, is not synchronized with system time; that is, the adjustment value is not used. When measuring short intervals of elapsed time, the difference between CPU time and system time is negligible. However, when measuring long intervals of elapsed time (such as several hours or more), the difference can be noticeable. Because of this possible “clock drift,” it is not recommended that you make just one call to the SIGNALTIMEOUT procedure to measure a long interval of elapsed time when you need a precise measurement that is synchronized with system time. Instead, you should use a sequence of two or more calls. The same applies to other procedures, such as DELAY, that also measure time by a CPU clock.

For example, suppose that you want your application to be notified at a specific system time, say 12:00 noon. Your program could compare Julian timestamps to compute the interval between the current system time and 12:00 noon, and then set a timer for that interval by calling the SIGNALTIMEOUT procedure. However, if 12:00 noon is several hours away, the timer might miss it by a noticeable amount because of clock drift. Instead, you could use the SIGNALTIMEOUT procedure to set a timer to expire shortly before 12:00 noon. When the timer expires (that is, when the Time signal message is delivered to $RECEIVE), your application could compute the remaining time (again, by comparing Julian timestamps), and then set another timer for the short interval that remains.

However, because the possibility of a discrepancy due to clock drift becomes greater as the interval being timed becomes longer, it is even safer to measure a long time interval by dividing it into a series of relatively short intervals. One method is to compute the interval between the current time and the desired time and set a timer to expire after half that interval. When the timer expires, compute the remaining time and set another timer to expire after half that interval, and so on, approaching the desired time by progressively smaller steps. The code example in the following subsection uses this method.

A Sample Long-Range Timer

The following program, written in C, allows the user to specify a time of day, according to system time, at which a timer will expire. The program uses Julian timestamps to compute the interval between the current system time and the desired system time, and then uses the SIGNALTIMEOUT procedure to set a timer to expire after half of that interval. After that expiration, the program computes the time remaining and sets another timer to expire after half that time, continuing this process reiteratively. The program is able to wait for as little as one interval unit (0.01 second) if the

JULIANTIMESTAMP procedure shows that the desired system time has not yet been reached.

The user can run the program with no parameters to see the current system time. For example,

```
10> RUN TIMER
```

causes the program to display the system time, in both GMT and LCT, and then to terminate. The user can set a timer by specifying a time of day in the form *`hour minute`*. A 24-hour clock is always used. For example, to specify the time 14:35, enter the command

```
11> RUN TIMER 14 35
```

If it is already past the specified time of day, the timer is set for that time on the following day. The program then displays the desired system time, in both GMT and LCT, and displays the number of interval units (0.01 second each) until the halfway point. At the halfway point, the program displays an indication that there has been a “timer pop” (timer expiration), displays the time of the timer pop, and displays the number of interval units until the next interim point. It continues in this manner until it indicates that the desired system time has been reached.

The `main()` function contains the main processing loop. It calls the `TargetGMT()` function to compute the GMT of the desired system time, and calls the `SetupTimeout()` function to implement the “series of intervals” algorithm. For each timer that is set, the `main()` function reads $RECEIVE to receive the timeout message and then calls the `SetupTimeout()` function again to check if the target has been reached and, if not, to set a timer for the next interval.

The code for the program follows:

```
#include <cextdecs>
#include <ktdmtyp.h>
#include <tal.h>
#include <stdio.h>
#include <stdlib.h>

#define   GET_GMT       0
#define   GMT_TO_LCT    0
#define   LCT_TO_GMT    2
#define   MY_NODE      -1
#define   ELAPSED_TIME_TIMEOUT   -22
#define   SYSTEM_MSG_RECVD        6
#define   MAX_TIMEOUT_CENTISECS   0x7FFFFFFF

void showTime(int64 jt, char *txt)
{
  int16 err;
  int16 DT[8];

  INTERPRETTIMESTAMP(jt,DT);
  printf("GMT: %d/%02d/%02d %02d:%02d:%02d.%03d%03d",
         DT[0],DT[1],DT[2],DT[3],DT[4],DT[5],DT[6],DT[7]);
  jt = CONVERTTIMESTAMP(jt,0,,&err);
  INTERPRETTIMESTAMP(jt,DT);
  if (err) printf(" *** error: %d",err);
  else
    printf("   LCT: %d/%02d/%02d %02d:%02d:%02d.%03d%03d",
           DT[0],DT[1],DT[2],DT[3],DT[4],DT[5],DT[6],DT[7]);
  printf("  %s\n",txt);
}



/*
 * Computes the Greenwich mean time (GMT) at the next local
 * civil time (LCT) occurrence of the specified hour and
 * minute. If the time has already passed today, the GMT is
 * computed for the target time tomorrow. Returns 0 if
 * successful.
 */
int TargetGMT(uint16 hour, uint16 minute, int64 *target )
{
  int16 err;
  int16 DateTime[8];
  int64 jts_current, jts_target;

  /* Get GMT at time of call; convert to LCT */

  jts_current = JULIANTIMESTAMP(GET_GMT);
  jts_current = CONVERTTIMESTAMP(jts_current, GMT_TO_LCT,
                                 MY_NODE, &err);
  if (err) return err;

  /*
   * Convert the LCT time to a Gregorian date and time. Then
   * adjust the fields of the Gregorian timestamp to get the
   * desired target time.
   */

  INTERPRETTIMESTAMP(jts_current, DateTime);
  DateTime[3] = hour;
  DateTime[4] = minute;
  DateTime[5] = DateTime[6] = DateTime[7] = 0;

\*
 * Convert the target time from Gregorian to Julian LCT.
 * If target time is before current time-of-day, add a
 * day's worth of microseconds to make the timer pop at the
 * target time tomorrow.
 *\

  jts_target = COMPUTETIMESTAMP(DateTime, &err);
  if (err) return -1;  /* bad hour or minute */

  if (jts_target < jts_current)
    jts_target += 24*3600*1000000LL;
  /* Convert target LCT to GMT */

  *target = CONVERTTIMESTAMP(jts_target, LCT_TO_GMT, MY_NODE,
                                                     &err);
  return err;
}



/*
 * Checks if the target has been reached. If not, sets a timer
 * for half the remaining interval. Returns 0 when target time
 * has been reached, -1 otherwise.
 *
 * N.B: This function rounds *up* the interval, so when the
 * target is reached it might overshoot up to 0.01 second.
 * Rounding *down* the interval will produce the opposite
 * effect--undershooting up to 0.01 second.
 */
int SetupTimeout(int64 jts_gmt_target)
{
  int64 jts_current, interval;

  jts_current = JULIANTIMESTAMP(GET_GMT);

  /*
   * Compute half of the interval, convert it from microseconds
   * to centiseconds (0.01 second each), and round up to the
   * closest centisecond.
   */

  interval = (jts_gmt_target - jts_current) / 2;
  interval = (interval + 9999) / 10000;

  if (interval <= 0)
    return 0;

  /*
   * The timeout value can be a maximum of (2^31 - 1)
   * centiseconds, or about 248 days.
   */

  if (interval > MAX_TIMEOUT_CENTISECS)
    interval = MAX_TIMEOUT_CENTISECS;

  printf("timeout for interval = %u\n\n", (int32) interval);
  SIGNALTIMEOUT((int32) interval);
  return -1;
}
```

```
void main(int argc, char **argv)
{
  int64 jts_gmt_target;
  int   business_day;
  short recv_num;
  int16 read_data;
  int hour, min;
  short err, last_err, status;

  if (argc < 2)
    { showTime(JULIANTIMESTAMP(0), "time now"); STOP(); }
  hour = atoi(argv[1]);
  min = atoi(argv[2]);

  /*
   * Get the file number for $RECEIVE. System messages
   * and waited I/O are enabled by default.
   */

  err = FILE_OPEN_("$RECEIVE", 8, &recv_num);

  TargetGMT(hour, min, &jts_gmt_target);
  showTime(jts_gmt_target, "target time");
  business_day = SetupTimeout(jts_gmt_target);

  while (business_day)
  {

   /*
    * Perform a waited read on $RECEIVE. For real
    * applications, nowaited reads should be used so that
    * processing can be done between the timer pops.
    */
    status = READX(recv_num, (char*) &read_data, 2);
    if (_status_gt(status))
     {
       FILE_GETINFO_(recv_num, &last_err);

       if ((last_err  == SYSTEM_MSG_RECVD)  &&
          (read_data == ELAPSED_TIME_TIMEOUT))
        {
          showTime(JULIANTIMESTAMP(0), "timer pop");
          business_day = SetupTimeout(jts_gmt_target);
        }
     }
  }
  showTime(JULIANTIMESTAMP(0), "day ended!");
}
```

Managing System Time

This subsection describes system-clock functions such as setting the system clock and checking the system clock.

Remember that system time is the result of periodically synchronizing the clocks in the system using a clock-averaging algorithm. By taking the average value of the various CPU clocks, the system creates the concept of system time. When you obtain the system time, you are really obtaining the time in the local CPU, corrected by an adjustment value that is periodically updated by the clock-averaging algorithm.

The system time contains four adjustments:

- Average time adjustment, to align the time on this processor with the average time. The operating system makes this adjustment automatically and periodically, by circulating a message among the processors.
- Average rate adjustment, to make the time on this processor advance at the same rate as the average time. The operating system makes this adjustment automatically from successive observations of the time adjustment.
- External time adjustment, to align the system time with an external reference source. This adjustment is specified by a call to the SYSTEMCLOCK_SET_ or SETSYSTEMCLOCK procedure.
- External rate adjustment, to make the system time advance at the same rate as the external source. This adjustment has been zero on all operating systems delivered since 2001, until the J06.14 and H06.25 RVUs. As of these RVUs, it is specified by a call to the SYSTEMCLOCK_SET_ or SETSYSTEMCLOCK procedure.

The operations you can perform on system time are:

- Check the system time (TIME procedure)
- Set or adjust the system time or system clock rate (SYSTEMCLOCK_SET_/SETSYSTEMCLOCK procedure)
- Add a transition to the daylight-saving-time (DST) transition table (ADDDSTTRANSITION procedure, which is superseded by the DST_TRANSITION_ADD_ procedure on G05.00 and later G-series release systems)
- You can perform the following operations on G05.00 and later G-series release systems:
    - Add a transition to the daylight-saving-time (DST) transition table (DST_TRANSITION_ADD_ procedure)
    - Delete a transition from the DST table (DST_TRANSITION_DELETE_ procedure)

- Modify an existing transition in the DST table (DST_TRANSITION_MODIFY_ procedure)
- Query a transition from the DST table (DST_GETINFO_ procedure)

The following paragraphs describe how to perform these operations.

Checking the System Clock

You can check the system time either by issuing the TIME command at the command-interpreter prompt or by calling the TIME procedure.

The TIME command displays the date and time on the terminal as follows:

```
1>  TIME
April 13, 1990 9:43:03
```

The TIME procedure returns the date and time in integer variables representing the year, month, day, hour, minute, second, and fraction of a second in 0.01-second units. For example:

```
INT DATE^AND^TIME[0:6];
 .
 .
CALL TIME(DATE^AND^TIME);
```

On return from the TIME procedure, DATE^AND^TIME contains the following information:

```
DATE^AND^TIME[0]   Gregorian year (for example, 1990)
DATE^AND^TIME[1]   Month of the year (1 to 12)
DATE^AND^TIME[2]   Day of the month (1 to 31)
DATE^AND^TIME[3]   Hour of the day (0 to 23)
DATE^AND^TIME[4]   Minute of the hour (0 to 59)
DATE^AND^TIME[5]   Second of the minute (0 to 59)
DATE^AND^TIME[6]   Hundredth of the second (0 to 99)
```

The time displayed by the TIME command or returned by the TIME procedure is the local civil time as given by the CPU in which the command or procedure runs. It is because of the clock-averaging algorithm discussed above that this value can be equated with system time.

Setting the System Clock

All RVUs support the SETSYSTEMCLOCK procedure. Beginning with the J06.14 and H06.25 RVUs, the SYSTEMCLOCK_SET_ procedure is supported, and called by the SETSYSTEMCLOCK procedure. The only difference between the old and new interface is the way errors are reported. SETSYSTEMCLOCK returns a condition code (< or = in TAL, a `_cc_status` value in C/C++); SYSTEMCLOCK_SET_ returns an integer, with a unique value for each error. However, for callers in native C/C++, the `_cc_status` value is actually an `int`; the same distinct error codes are available through SETSYSTEMCLOCK. Only TNS or pTAL programs need to call SYSTEMCLOCK_SET_ explicitly to see the distinct error results. For details of the error codes, see SYSTEMCLOCK_SET_ in the *Guardian Procedure Calls Reference Manual*. SETSYSTEMCLOCK has the advantage that it is implemented in all RVUs; on any RVU that supports both procedures, it accepts all the same parameter values as SYSTEMCLOCK_SET_, so it can be used with identical effect.

You can set the system clock either programmatically using the SETSYSTEMCLOCK procedure or interactively from the TACL prompt using the SETTIME command. TACL also has a built-in function, #SETSYSTEMCLOCK, to call the SETSYSTEMCLOCK procedure, but it accepts only mode values 0 through 3. A utility program like the example below can be used to invoke SETSYTEMCLOCK interactively. You must have an ID in the SUPER.* group (group ID = 255) to use either the SETSYSTEMCLOCK procedure or the SETTIME command.

Using the SYSTEMCLOCK_SET_ or SETSYSTEMCLOCK Procedure

You typically use the SETSYSTEMCLOCK procedure to synchronize the system clock with an external clock. To provide a timestamp with finer tolerance, you can connect an external clock to your system, typically using the Network Time Protocol (NTP) or Simple NTP (SNTP). You need to regularly compare the timestamps issued on your own system with a timestamp issued by the external clock. System action depends on the value of the *mode* parameter along with the amount that the call to SETSYSTEMCLOCK intends to change the time.

All systems support nine modes for SETSYSTEMCLOCK to adjust or set the system time to an absolute value or by a relative value. As of the J06.14 and H06.25 RVUs, SYSTEMCLOCK_SET_ and SETSYSTEMCLOCK support two additional modes to adjust the clock rate (frequency). For details, see the *Guardian Procedure Calls Reference Manual*.

The SCLOCK program distributed with your system is an example of a program that synchronizes your system clock with an external clock connected by an asynchronous line. This program uses the SETSYSTEMCLOCK procedure to adjust the system clock.

The following example is a simple utility program to call SETSYSTEMCLOCK interactively. On J06.14, H06.25, and subsequent RVUs, it reports distinct negative integers for each error; on earlier RVUs all errors are reported as –1. The most useful mode for making gradual adjustments to the system time is mode 6. Mode 5 is useful to correct the time immediately, but should be used only when no software is running that is sensitive to abrupt or negative changes in successive timestamp values. As of J06.14 and H06.25, mode 9 can be used to adjust the clock rate.

```
#include <stdio.h> nolist
#include <stdlib.h> nolist
#include <errno.h>
#include <cextdecs(SETSYSTEMCLOCK)>
int main (int argc, char *argv[])
{
   short mode, tuid;
   unsigned long t;
   long long JulianGMT;
   char *p;
   int result;
   puts("SSCK: SETSYSTEMCLOCK utility\n"
         "(c) Copyright 2012 Hewlett-Packard Development "
         "Company, L.P.\n");
   if (argc < 2 || argc > 4) {
      puts("Provide one to three parameters:\n"
            "  1st is mode (short, required)\n"
            "  2nd is 'julianGMT' (long long, depends on mode, "
               "sometimes opt)\n"
            "  3rd is TUID (short, optional)");
      return (argc > 4);
   }
   t = strtoul(argv[1], &p, 10);
   if (*p || t == 0 && errno || t > 999)
       return printf("Invalid mode: %s\n", argv[1]), 1;
   mode = (short)t;
   if (argc > 2) {
       JulianGMT = strtoll(argv[2], &p, 10);
       if (*p || JulianGMT == 0 & errno)
           return printf("Invalid JulianGMT: %s\n", argv[2]), 1;
       if (argc > 3) {
           t = strtoul(argv[3], &p, 10);
           if (*p || t == 0 && errno || t > 65535)
               return printf("Invalid TUID: %s\n", argv[3]), 1;
           tuid = (short)t;
        }
   }
   printf("SETSYSTEMCLOCK(");
   if (argc > 2) printf("%lld", JulianGMT);
   printf(", %d", mode);
   if (argc > 3) printf(", %d", tuid);
   result = SETSYSTEMCLOCK(_optional(argc > 2, JulianGMT),
                           mode,
                           _optional(argc > 3, tuid));
   printf(") => %d\n", result);
}
```

This example can be copied to a file named `ssckc` and compiled with TNS/E native C as follows:

```
ccomp /in ssckc/ssck;suppress,extensions,symbols,runnable
```

Using the SETTIME Command

You rarely need to use the SETTIME command. For most systems, you need to use SETTIME only when the system is first cold loaded. Only on older hardware is it necessary to reset the system time following each cold load or power failure.

Interacting With the DST Transition Table

The daylight-saving-time (DST) transition table provides a way of indicating the time and date at which transitions to and from daylight saving time will be made. On D-series systems, you can set the DAYLIGHT_SAVING_TIME entry in the configuration file to NONE, USA66, or TABLE during system generation. See the *System Generation Manual* for details about the configuration file.

You can use the Subsystem Control Facility (SCF) to change the DAYLIGHT_SAVING_TIME setting on your machine. The “SCF info subsys $zzkrn" command shows the current setting for DAYLIGHT_SAVING_TIME (as well as the TIME_ZONE_OFFSET). The "SCF alter subsys $zzkrn, DAYLIGHT_SAVING_TIME TABLE" command changes the setting to TABLE. The change will take effect only at the next system cold load. See the *SCF Reference Manual for the Kernel Subsystem* for more information.

The following shows the result of setting the DAYLIGHT_SAVING_TIME entry to one of the three available options:

NONE	DST does not apply.
USA66	The system automatically follows the rules for daylight-saving-time set for the United States by the Uniform Time Act of 1966.
TABLE	You need to put entries into the DST transition table.

For G04.00 and earlier G-series and D-series releases of the system, you can fill out the DST transition table either interactively using the ADDDSTTRANSITION TACL command, or programmatically using the ADDDSTTRANSITION procedure. You must be a super-group user (255, *n*) to set information in the DST transition table.

For G05.00 and later G-series release systems, you can also delete an entry, modify an entry, or get information about an entry in the DST table using the following procedures: DST_TRANSITION_DELETE_, DST_TRANSITION_MODIFY_ and DST_GETINFO_. For a description of the various error values returned by these procedures, refer to the *Guardian Procedure Calls Reference Manual*. DST_TRANSITION_ADD_ supersedes the ADDDSTTRANSITION procedure.

You must a super-group user(255, *n*) to use ADDDSTTRANSITION, DST_TRANSITION_ADD_, DST_TRANSITION_DELETE_ or DST_TRANSITION_MODIFY_.

If you choose to use the TABLE option, you must consider the following:

- You must have at least one transition date that is less than the current date and time, and at least two transition dates that are later than the current date and time.
- Your first DST transition must be earlier than any date that will be referenced by the BACKUP utility, any other utility, or application program.

Using the ADDDSTTRANSITION Procedure

You supply the ADDDSTTRANSITION procedure with two Julian timestamps and an offset. The timestamps specify the beginning and the end of a time period. The offset specifies the number of seconds that the LCT offsets the LST for the specified period. The following example sets three transitions.

Note that successive calls to ADDDSTTRANSITION must load the DST table in time sequence with no gaps. Every call to ADDDSTTRANSITION except the first call must have the *`low-gmt`* parameter equal to the *`high-gmt`* parameter of the previous call. These restrictions do not apply on G05.00 and later G-series release systems.

```
!First DST period; April 14, 1991 through October 20, 1991:

DATE^AND^TIME[0] := 1991;         !year
DATE^AND^TIME[1] := 4;            !month
DATE^AND^TIME[2] := 14;           !day
DATE^AND^TIME[3] := 2;            !hour
DATE^AND^TIME[4] := 0;            !minute
DATE^AND^TIME[5] := 0;            !second
DATE^AND^TIME[6] := 0;            !millisecond
DATE^AND^TIME[7] := 0;            !microsecond
LOW^GMT := COMPUTETIMESTAMP(DATE^AND^TIME,
                            ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL INVALID^DATE;

DATE^AND^TIME[0] := 1991;         !year
DATE^AND^TIME[1] := 10;           !month
DATE^AND^TIME[2] := 20;           !day
DATE^AND^TIME[3] := 2;            !hour
DATE^AND^TIME[4] := 0;            !minute
DATE^AND^TIME[5] := 0;            !second
DATE^AND^TIME[6] := 0;            !millisecond
DATE^AND^TIME[7] := 0;            !microsecond
HIGH^GMT := COMPUTETIMESTAMP(DATE^AND^TIME,
                             ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL INVALID^DATE;

OFFSET := SECONDS^IN^HOUR;
CALL ADDDSTTRANSITION(LOW^GMT,HIGH^GMT,OFFSET);

!Second DST period; October 20, 1991 through April 12, 1992:

LOW^GMT := HIGH^GMT;
DATE^AND^TIME[0] := 1992;         !year
DATE^AND^TIME[1] := 4;            !month
DATE^AND^TIME[2] := 12;           !day
DATE^AND^TIME[3] := 2;            !hour
DATE^AND^TIME[4] := 0;            !minute
DATE^AND^TIME[5] := 0;            !second
DATE^AND^TIME[6] := 0;            !millisecond
DATE^AND^TIME[7] := 0;            !microsecond

HIGH^GMT := COMPUTETIMESTAMP(DATE^AND^TIME,
                             ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL INVALID^DATE;

OFFSET := 0;
CALL ADDDSTTRANSITION(LOW^GMT,HIGH^GMT,OFFSET);

!Third DST period; April 12, 1992 through October 18, 1992:

LOW^GMT := HIGH^GMT;
DATE^AND^TIME[0] := 1992;       !year
DATE^AND^TIME[1] := 10;         !month
DATE^AND^TIME[2] := 18;         !day
DATE^AND^TIME[3] := 2;          !hour
DATE^AND^TIME[4] := 0;          !minute
DATE^AND^TIME[5] := 0;          !second
DATE^AND^TIME[6] := 0;          !millisecond
DATE^AND^TIME[7] := 0;          !microsecond
HIGH^GMT := COMPUTETIMESTAMP(DATE^AND^TIME,
                             ERROR^MASK);
IF ERROR^MASK <> 0 THEN CALL INVALID^DATE;

OFFSET := SECONDS^IN^HOUR;
CALL ADDDSTTRANSITION(LOW^GMT,HIGH^GMT,OFFSET);
 .
 .
```

Note that the second call to ADDDSTTRANSITION, where zero offset is specified, is not required on G05.00 and later G-series systems.

Using the ADDDSTTRANSITION Command

The ADDDSTTRANSITION TACL command has the same effect as calling the ADDDSTTRANSITION or the DST_TRANSITION_ADD_ procedures. Again you supply two Julian timestamps that mark the beginning and end of the period and an offset in hours and minutes. The following example uses ADDDSTTRANSITION commands to achieve the same effect as the ADDDSTTRANSITION procedure calls in the previous example:

```
1> ADDDSTTRANSITION 14 APR 1991, 2:00 GMT, 20 OCT 1991, 2:00 GMT, 1:00
2> ADDDSTTRANSITION 20 OCT 1991, 2:00 GMT, 12 APR 1992, 2:00 GMT, 0:00
3> ADDDSTTRANSITION 12 APR 1992, 2:00 GMT, 18 OCT 1992, 2:00 GMT, 1:00
```

Note that the second ADDDSTTRANSITION command is not necessary on G05.00 and later G-series systems, because zero offset is specified. Also, note that the ADDDSTTRANSITION command allows you to specify LST instead of GMT.

Using the DST_TRANSITION_ADD_ Procedure

You supply the DST_TRANSITION_ADD_ procedure with a pointer to the zsys_ddl_dst_entry_def structure with its fields filled in. The z_lowgmt and z_highgmt fields are Julian timestamps that specify the beginning and the end of a time period, respectively. The z_offset field specifies the number of seconds that the LCT offsets the LST for the specified time period. The z_version field must be set to DST_VERSION_SEP1997.

Note the following rules when adding entries to the DST table:

1. The z_lowgmt and z_highgmt fields must have values between 1/1/1 and 12/31/10000.
2. There must be no existing entry in the DST table for which both of the following are true:
    - The entry has a nonzero offset
    - The entry overlaps the time period bounded by z_lowgmt and z_highgmt fields.
3. The DST table stores entries with nonzero offset. The entries with zero offset are deduced from the gaps in the table. Hence, if z_offset is zero and rules (1) and (2) are satisfied, the operation does not affect the contents of the table. This means that only entries with nonzero offset need to be added to the table.

```
#include <zsysc>

#include <cextdecs(COMPUTETIMESTAMP,DST_TRANSITION_ADD_)>

zsys_ddl_dst_entry_def DSTEntry;
short error, dateAndTime[8], errorMask;
long long timeStampLow, timeStampHigh;

/* First DST period; April 14, 1991 through October 20, 1991,
   Offset = 1 hour */

dateAndTime[0] = 1991; /* year */
dateAndTime[1] = 4;    /* month */
dateAndTime[2] = 14;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampLow = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

dateAndTime[0] = 1991; /* year */
dateAndTime[1] = 10;   /* month */
dateAndTime[2] = 20;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampHigh  = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

DSTEntry.z_lowgmt = timeStampLow;
DSTEntry.z_highgmt = timeStampHigh;
DSTEntry.z_offset = 3600; /* seconds in 1 hour */
DSTEntry.z_version = DST_VERSION_SEP1997;
error = DST_TRANSITION_ADD_(&DSTEntry);
```

```
if (error != ZSYS_VAL_DST_OK) errorExit();

/* Second DST period; October 20, 1991 through April 12, 1992,
   Offset = 0 */
/* Since Offset = 0, there is no need to explicitly add this
   entry.*/
/* Third DST period; April 12, 1992 through October 18, 1992,
   Offset = 1 hour */
dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 4;    /* month */
dateAndTime[2] = 12;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampLow = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 10;   /* month */
dateAndTime[2] = 18;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampHigh  = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

DSTEntry.z_lowgmt = timeStampLow;
DSTEntry.z_highgmt = timeStampHigh;
DSTEntry.z_offset = 3600; /* seconds in 1 hour */
DSTEntry.z_version = DST_VERSION_SEP1997;
error = DST_TRANSITION_ADD_(&DSTEntry);

if (error != ZSYS_VAL_DST_OK) errorExit();
.
.
.
.
```

Using the DST_TRANSITION_DELETE_ Procedure

You supply the DST_TRANSITION_DELETE_ procedure with a pointer to the zsys_ddl_dst_entry_def structure with its fields filled in. The fields describe an existing entry in the DST table.

The following rules have to be kept in mind while deleting entries from the DST table:

1. Only transitions that already exist in the table can be deleted. Deleting an entry that has a zero offset has no effect and the table remains unaltered.
2. An attempt to delete the entry that is currently in effect is not allowed when the offset field of that entry has a nonzero value. The DST_TRANSITION_MODIFY_ procedure may be used to delete such an entry. See rule (2) of Using the DST_TRANSITION_MODIFY_ Procedure.

```
#include <zsysc>

#include <cextdecs(COMPUTETIMESTAMP,DST_TRANSITION_DELETE_)>

zsys_ddl_dst_entry_def DSTEntry;

short error, dateAndTime[8], errorMask;

long long timeStampLow, timeStampHigh;

/* Delete the third transition added by the DST_TRANSITION_ADD_
   procedure above */

dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 4;    /* month */
dateAndTime[2] = 12;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampLow = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 10;    /* month */
dateAndTime[2] = 18;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampHigh  = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

DSTEntry.z_lowgmt = timeStampLow;
DSTEntry.z_highgmt = timeStampHigh;
DSTEntry.z_offset = 3600; /* seconds in 1 hour */
DSTEntry.z_version = DST_VERSION_SEP1997;
error = DST_TRANSITION_DELETE_(&DSTEntry);
if (error != ZSYS_VAL_DST_OK) errorExit();
.
.
.
.
```

Using the DST_TRANSITION_MODIFY_ Procedure

You supply the DST_TRANSITION_MODIFY_ procedure with pointers to two zsys_ddl_dst_entry_def structures with their fields filled in. The first of these structures describes an existing entry in the DST table that has to be modified. The second structure describes a new entry that will replace the entry that needs to be modified.

The following rules have to be kept in mind while modifying entries from the DST table:

1. Existing transitions with nonzero offsets can be modified if the new values will not overlap other existing transitions that have nonzero offsets.
2. Calling the DST_TRANSITION_MODIFY_ procedure with the z_offset field set to zero in the second structure deletes the entry described by the first structure.

WARNING. If the entry that is currently in effect is modified and the offset value is changed, then you should be aware that there will be jumps in the Local Civil Time. If your applications cannot tolerate such jumps, then you should not attempt to modify the entry that is currently in effect.

```
#include <zsysc>

#include <cextdecs(COMPUTETIMESTAMP,DST_TRANSITION_MODIFY_)>

zsys_ddl_dst_entry_def oldDSTEntry, newDSTEntry;

short error, dateAndTime[8], errorMask;

long long timeStampLow, timeStampHigh;

/* Modify the third transition added by the DST_TRANSITION_ADD_
   procedure above */

dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 4;    /* month */
dateAndTime[2] = 12;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
dateAndTime[7] = 0;    /* microsecond */
timeStampLow = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

dateAndTime[0] = 1992; /* year */
dateAndTime[1] = 10;    /* month */
dateAndTime[2] = 18;   /* day */
dateAndTime[3] = 2;    /* hour */
dateAndTime[4] = 0;    /* minute */
dateAndTime[5] = 0;    /* second */
dateAndTime[6] = 0;    /* millisecond */
```

```
dateAndTime[7] = 0;    /* microsecond */
timeStampHigh  = COMPUTETIMESTAMP(dateAndTime, &errorMask);

if (errorMask != 0) errorExit();

oldDSTEntry.z_lowgmt = timeStampLow;
oldDSTEntry.z_highgmt = timeStampHigh;

oldDSTEntry.z_offset = 3600; /* seconds in 1 hour */
oldDSTEntry.z_version = DST_VERSION_SEP1997;
newDSTEntry.z_lowgmt = timeStampLow;
newDSTEntry.z_highgmt = timeStampHigh;
newDSTEntry.z_offset = 7200; /* seconds in 2 hours */
newDSTEntry.z_version = DST_VERSION_SEP1997;
error = DST_TRANSITION_MODIFY_(&oldDSTEntry, &newDSTEntry);

if (error != ZSYS_VAL_DST_OK) errorExit();
.
.
.
.
```

Using the DST_GETINFO_ Procedure

You supply a Julian timestamp and a pointer to the zsys_ddl_dst_entry_def structure. DST_GETINFO_ fills in the fields of the zsys_ddl_dst_entry_def structure with information about the DST entry that was, is, or will be in effect at the time specified by the Julian timestamp.

```
#include <zsysc>

#include <cextdecs(COMPUTETIMESTAMP,DST_GETINFO_)>

zsys_ddl_dst_entry_def oldDSTEntry, newDSTEntry;

short error, dateAndTime[8], errorMask;

long long timeStampLow, timeStampHigh;

/* Use the DST_GETINFO_ procedure to print the contents of the
DST
   transition table */

DSTEntry.z_version = ZSYS_VAL_DST_VERSION_SEP1997;

/* Calling DST_GETINFO_ with -1 for timestamp returns the first
   nonzero DST transition */

error = DST_GETINFO_(-1, &DSTEntry);
while (error == 0)
{
    printDSTEntry(&DSTEntry);
    error = DST_GETINFO_(DSTEntry.z_highgmt, &DSTEntry);
}
```

19

Formatting and Manipulating Character Data

This section describes how to use the character formatting and editing capabilities of the operating system. Included here are discussions of the following:

- How to use the formatter (FORMATCONVERT[X] and FORMATDATA[X] procedures) for presenting data in an organized way, such as for displaying tabulated data. Using the Formatter provides details.
- How to perform operations on character strings such as changing the case of alphabetic characters (SHIFTSTRING procedure), converting numeric data between ASCII representation and binary numbers (NUMIN, NUMOUT, DNUMIN, and DNUMOUT procedures), editing a character string (FIXSTRING procedure), or sorting characters (HEAPSORT[X_] procedure). See Manipulating Character Strings for details.
- How to manipulate multibyte character sets such as those used for representing the Japanese, Chinese, and Korean languages (MBCS_* procedures). See Programming With Multibyte Character Sets for details.

Using the Formatter

The formatter enables you to arrange lists of data items on output or input. The way you arrange data can be format-directed or list-directed:

- Format-directed formatting arranges data items according to a sequence of edit descriptors that specify a format. Using the edit descriptors, you can specify how and where data items are displayed and you can specify the data type; the system will do any necessary conversion for you (such as converting numeric data into ASCII). Format-directed formatting is used mostly in formatting data on output to display it in a readable way.
- List-directed formatting does not use a specified format but formats data using data-type information that is entered as an attribute of the data item. This method is less powerful than format-directed formatting for arranging data. Its major use is in interpreting free-format input and then storing that input in a compact form.

This subsection discusses format-directed formatting and list-directed formatting and describes several of the more common formatting tasks that you can perform. Specifically, it discusses the FORMATCONVERT[X] and FORMATDATA[X] procedures, which perform the formatting.

The FORMATCONVERT and FORMATCONVERTX procedures are identical except that FORMATCONVERT requires that all of its reference parameters be 16-bit addresses, while FORMATCONVERTX accepts extended (32-bit) addresses for all of its reference parameters.

The FORMATDATA and FORMATDATAX procedures are also identical except that FORMATDATA requires that all of its reference parameters be 16-bit addresses, while FORMATDATAX accepts extended (32-bit) addresses for all of its reference parameters.

The FORMATCONVERT procedure is used in combination with FORMATDATA, while the FORMATCONVERTX conversion is used in combination with FORMATDATAX.

Note. TNS/R native programs that perform formatting must use FORMATCONVERTX and FORMATDATAX rather than FORMATCONVERT and FORMATDATA.

The direction of the format conversion—format directed or list directed, input or output—is determined by the `flags` parameter passed to the FORMATDATA[X] procedure.

This subsection does not describe every available edit descriptor, nor does it describe all aspects of every edit descriptor that it mentions. For complete details on all edit descriptors, refer to the *Guardian Procedure Calls Reference Manual*.

Format-Directed Formatting

Format-directed formatting works by providing the FORMATDATA[X] procedure with a sequence of edit descriptors that specify how data is to be formatted. You specify format-directed formatting by setting bit 2 of the `flags` parameter to zero (the default value) when calling the FORMATDATA[X] procedure.

You can apply a format to output data or input data as follows:

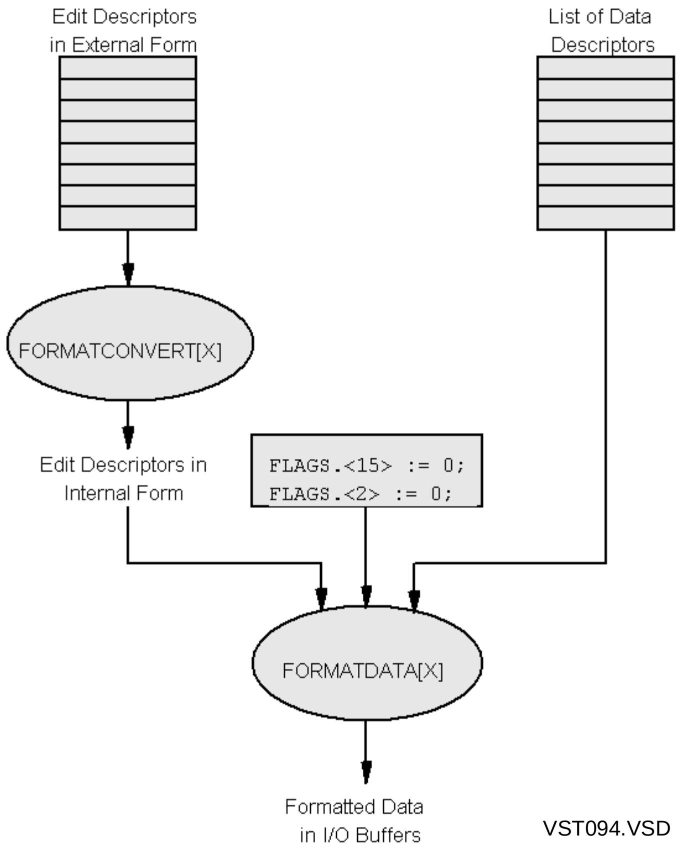
- When formatting output, you supply the data to be formatted in an internal form. The FORMATDATA[X] procedure converts the data into an external form according to the specified format.
- When formatting input, you supply data in external form. The FORMATDATA[X] procedure converts the input into internal form according to the specified format.

The following paragraphs describe output formatting and input formatting in detail and discuss some of the more common formatting operations.

Formatting Output

Figure 19-1 shows the role of the FORMATCONVERT[X] and FORMATDATA[X] procedures in formatting data for output according to a specified format.

Figure 19-1. Format-Directed Formatting


Edit Descriptors in External Form
List of Data Descriptors
FORMATCONVERT[X]
Edit Descriptors in Internal Form
FLAGS.<15> := 0;
FLAGS.<2> := 0;
FORMATDATA[X]
Formatted Data in I/O Buffers
VST094.VSD

Setting bit 15 of the *`flags`* parameter to zero specifies that the FORMATDATA[X] procedure will perform output formatting. Setting bit 2 to zero specifies format-directed formatting.

You provide the format as a series of edit descriptors in external form (as an ASCII string) to the FORMATCONVERT[X] procedure. This procedure converts the edit descriptors into an internal form understood by the FORMATDATA[X] procedure.

The list of data descriptors describes the data to be converted. Each data descriptor is made up of an array that describes one data item or sequence of data items that corresponds to one edit descriptor.

The FORMATDATA[X] procedure takes the data items pointed to by the data descriptors and formats them according to the internal format provided by the FORMATCONVERT[X] procedure. FORMATDATA[X] places the output in the I/O buffers.

The FORMATDATA[X] procedure reads the edit descriptors from left to right and retrieves data descriptors from the top of the data descriptor list when required by the edit descriptor. Note that while every data descriptor has a corresponding edit descriptor, not every edit descriptor has a corresponding data descriptor. Some edit descriptors, for example, provide tabulation information and therefore move a pointer to a specific location without accessing any data.

Formatting Input

Figure 19-2 shows the role of the FORMATCONVERT[X] and FORMATDATA[X] procedures in formatting input data according to a specified format.

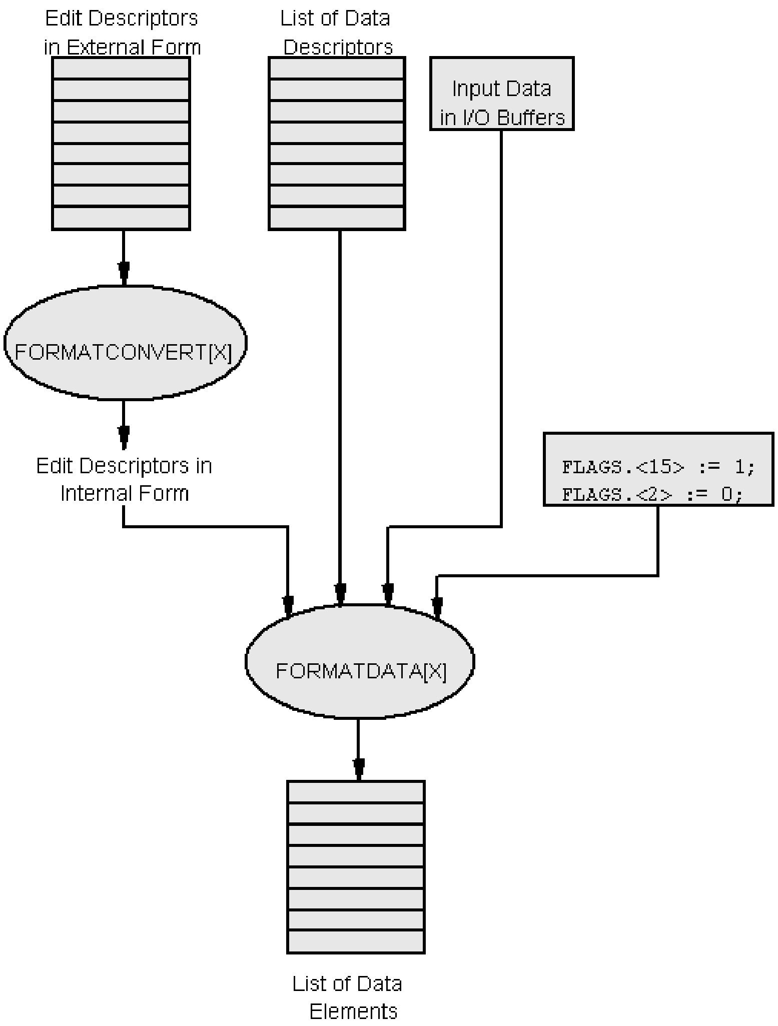
For input formatting, bit 15 of the flags parameter supplied to the FORMATDATA[X] procedure must be set to 1. Bit 2 is set to zero for format-directed formatting.

Again, the format is specified as a sequence of edit descriptors that you supply to the FORMATCONVERT[X] procedure in external form. The FORMATCONVERT[X] procedure converts the input string into an internal form suitable for passing to the FORMATDATA[X] procedure.

You supply the input data in the I/O buffer (typically in ASCII code). The list of data descriptors describes the placeholders that will contain the internal form of the data when it has been converted by the FORMATDATA[X] procedure.

The FORMATDATA[X] procedure uses the format supplied by the FORMATCONVERT[X] procedure to format the data supplied in the I/O buffer. The formatted data gets stored in the variables as described by the list of data descriptors. Note that the list of data elements is actually unchanged on output, but the variables now contain formatted data.

Figure 19-2. Formatting Input


Edit Descriptors in External Form
List of Data Descriptors
Input Data in I/O Buffers
FORMATCONVERT[X]
Edit Descriptors in Internal Form
FLAGS.<15> := 1;
FLAGS.<2> := 0;
FORMATDATA[X]
List of Data Elements

VST095.VSD

Introduction to Edit Descriptors

As indicated previously, for both input formatting and output formatting you need to supply the FORMATCONVERT[X] procedure with a sequence of edit descriptors that specify how the FORMATDATA[X] procedure will format the data. This sequence of edit descriptors must be supplied in an external (ASCII) format to FORMATCONVERT[X].

The external format consists of a sequence of edit descriptors separated by commas. Edit descriptors can be repeatable or nonrepeatable as described below:

- Repeatable edit descriptors include all edit descriptors that require data. Repeatable edit descriptors include all kinds of numbers and ASCII characters. They are called repeatable because the edit descriptor can specify multiple occurrences of the data type. The corresponding data descriptor must point to an array of multiple data elements that will satisfy the repeated edit descriptor.

  The following are examples of repeatable edit descriptors:

I5	A five-numeric integer
10(A12)	A 12-character alphanumeric string repeated 10 times

- Nonrepeatable edit descriptors do not correspond to data. They contain all the information needed for formatting. Nonrepeatable edit descriptors include literals, tabulation descriptors, and buffer-control characters. They are called nonrepeatable edit descriptors because one edit descriptor cannot represent multiple data items.

  The following are examples of nonrepeatable edit descriptors:

TR8	Moves the buffer pointer eight character positions to the right of the current position
EIGHT	The literal “EIGHT”

Each edit descriptor can have its properties changed using special character sequences called modifiers or decorations:

- A modifier is a code used to alter the results of the formatting prescribed by the edit descriptor to which it belongs. Modifiers include left and right justification and fill-character specification.

  The following example uses the “LJ” modifier to left-justify a 12-character string:

  ```
  [LJ]A12
  ```

- A decoration specifies alphanumeric strings that can be added to a field either before formatting begins or after it has finished.

  The following example uses the “MA” decoration to print the text “negative number” before a fixed-point number if the number is negative:

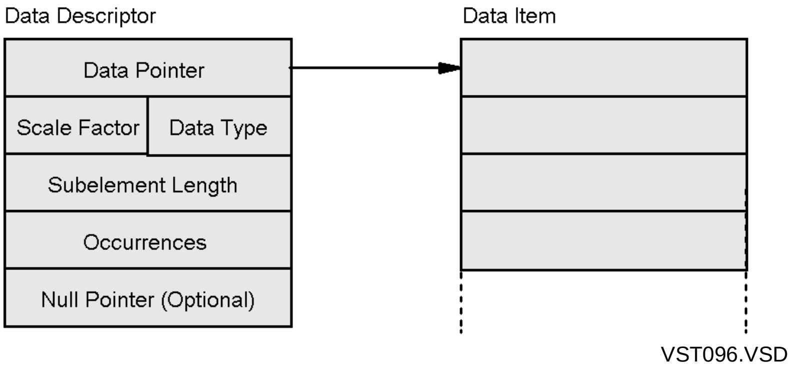
  ```
  [MA"negative number"]F10.3
  ```

Several examples of common uses of edit descriptors are given throughout the remainder of this subsection. For a description of every edit descriptor, modifier, and decoration, see the *Guardian Procedure Calls Reference Manual*.

Introduction to Data Descriptors

Each data descriptor describes an internal data item as shown in Figure 19-3:

Figure 19-3. Contents of a Data Descriptor



The data pointer points to the item of data. The data pointer is one word if the data item is in the user data segment or two words if the data item is in an extended data segment.

The scale factor is one byte long and is normally zero, but it can adjust the position of the implied decimal point: a positive value moves the implied decimal point to the right; a negative number moves it to the left.

The data type is also one byte long and indicates the type of data that the data item contains; for example, the data type for a string is 0 and the data type for a signed integer is 2. For a complete list of data types, see the FORMATDATA[X] procedure in the *Guardian Procedure Calls Reference Manual*.

The subelement length gives the length in bytes of each element in the data item. For example, if the corresponding edit descriptor specifies a six-character text item, then the subelement length is 6.

The number of occurrences indicates the number of repetitions of the element in the data item. For example, if the corresponding edit descriptor specifies a six-character text item repeated 10 times, then the number of occurrences is 10. In this case, the actual length of the data item is 60 characters.

The null pointer is the byte address of the null value if used. If the data item is in the user data segment, then this value is one word in length. If the data item is in an extended data segment, then this value is two words long.

Formatting Numbers, Text, and Other Data Items

There are several edit descriptors that you can use to process and format data for output:

A	Formats ASCII-coded text; the input is usually a string type but could also be numeric—the binary numbers are interpreted as ASCII characters.
B	Converts a number from its internal representation into ASCII code for output as a binary number according to a specified format.
D	Is identical to the E edit descriptor.
E	Converts a binary floating-point number into ASCII code for output as a decimal number according to a specified format.
F	Converts a binary fixed-point number into ASCII code for output as a decimal number according to a specified format.
G	Converts a binary fixed-point or floating-point number into ASCII code for output as a decimal number according to a specified format.
I	Converts a binary number into an integer for output according to a specified format. The output is in ASCII code and can be in any specified numeric base from 2 to 16, inclusive.
L	Processes the input value and returns a true or false indication: "T" if the value is nonzero, "F" if the value is zero.
M	Edits alphanumeric or numeric data according to an editing pattern or mask.
O	Converts a number from its internal representation into ASCII code for output as an octal number according to a specified format.
Z	Converts a number from its internal representation into ASCII code for output as a hexadecimal number according to a specified format.

The edit descriptors listed above are known as repeatable edit descriptors because the edit descriptor can be applied repeatedly to several data items in an array pointed to by the list data element. To repeat an edit descriptor, you enclose the edit descriptor in parentheses and precede it with a number indicating the number of repetitions. The following example reserves six character positions on output for a logical data item and repeats three times:

```
3(L6)
```

In this case, the corresponding data descriptor points to an array with at least three entries in it. The output indicates logical values for the first three values in the array.

For example:

```
Array values: 27, 6789.3, 0
Output: "      T      T      F"
```

The code fragment shown below processes some alphanumeric characters and some numeric characters.

The first part of the code fragment sets up the edit descriptors for seven alphanumeric data items to be retrieved using the first data descriptor and displayed with five characters each, and for seven integer data items to be retrieved from the array pointed to by the second data descriptor and displayed in five character positions each. The FORMATCONVERT procedure returns an internal form of the edit descriptors in the IFORMAT variable.

The code fragment sets up two arrays: one to contain the seven items of alphanumeric data (DAYS^ARRAY) and one to contain numeric data (INT^ARRAY). Two data descriptors point to these arrays: VLIST[0] points to DAYS^ARRAY, and VLIST[1] points to INT^ARRAY. In addition to the pointers, these data descriptors also indicate the scale factor, the size of each data element, and the number of occurrences.

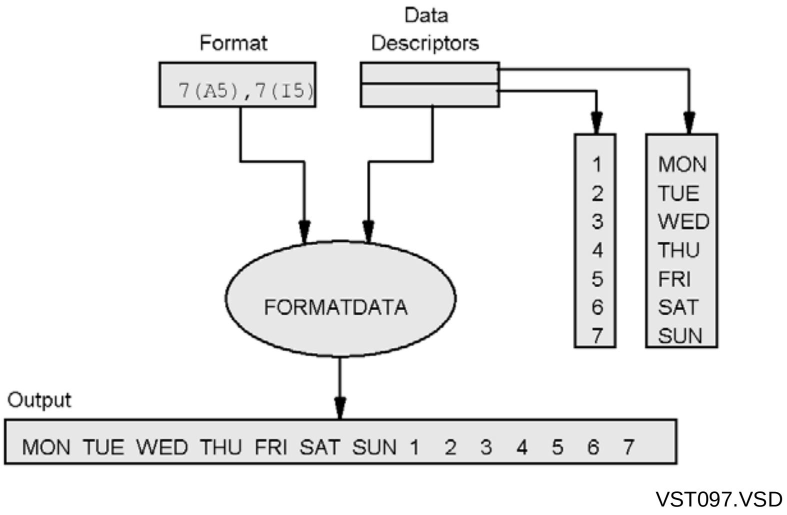
Finally, the code fragment calls the FORMATDATA procedure. The major input parameters to this procedure are the data descriptors in the VLIST array and the internal format in WFORMAT. Note that WFORMAT is a word pointer to the string array returned by the FORMATCONVERT procedure in IFORMAT.

```
!Set up the edit descriptors and convert to internal form:
EFORMAT ':=' "7(A5),7(I5)";
SCALES := 0;
CONVERSION := 1;
ERROR := FORMATCONVERT(IFORMAT,
                       IFORMATLEN,
                       EFORMAT,
                       EFORMATLEN,
                       SCALES,
                       SCALE^COUNT,
                       CONVERSION);
IF ERROR <= 0 THEN ...

!Set up arrays for the days of the week and the date:
DAYS^ARRAY ':=' ["MON","TUE","WED","THU","FRI","SAT","SUN"];
INT^ARRAY ':=' [1,2,3,4,5,6,7];

!Set up list elements that point to the above arrays:
VLIST^LEN := 2;
FLAGS := 0;
VLIST[0].ELEMENT^PTR := @DAYS^ARRAY;
VLIST[0].ELEMENT^SCALE := 0;
VLIST[0].ELEMENT^TYPE := 0;
VLIST[0].ELEMENT^LENGTH := 4;
VLIST[0].ELEMENT^OCCURS := 1;
VLIST[1].ELEMENT^PTR := @INT^ARRAY;
VLIST[1].ELEMENT^SCALE := 0;
VLIST[1].ELEMENT^TYPE := 2;
VLIST[1].ELEMENT^LENGTH := 2;
VLIST[1].ELEMENT^OCCURS := 7;
```

```
!Format the data:
ERROR := FORMATDATA(BUFFERS,
                    BUFFER^LENGTH,
                    NUM^BUFFERS,
                    BUFFER^ELEMENTS,
                    WFORMAT,
                    VLIST,
                    VLIST^LEN,
                    FLAGS);
IF ERROR <> 0 THEN ...
```

Figure 19-4 shows the operation of the FORMATDATA procedure for this example. (For simplicity the FORMATCONVERT procedure is omitted from this figure.)

Figure 19-4. Formatting Numbers and Text


Data
Descriptors
Format
7(A5),7(I5)
1
2
3
4
5
6
7
MON
TUE
WED
THU
FRI
SAT
SUN
FORMATDATA
Output
MON TUE WED THU FRI SAT SUN 1 2 3 4 5 6 7
VST097.VSD

Using Buffer Control

It is often convenient to use multiple buffers for output from the FORMATDATA[X] procedure. In addition to making it easier to handle larger amounts of output data, multiple buffers also help format data into lines for output, because you can then issue one WRITE procedure call for each buffer.

To terminate a buffer and start a new one, you put a slash (/) character in the edit descriptor string. When using multiple buffers, the `buffer` parameter to the FORMATDATA[X] procedure must identify a series of contiguous buffers.

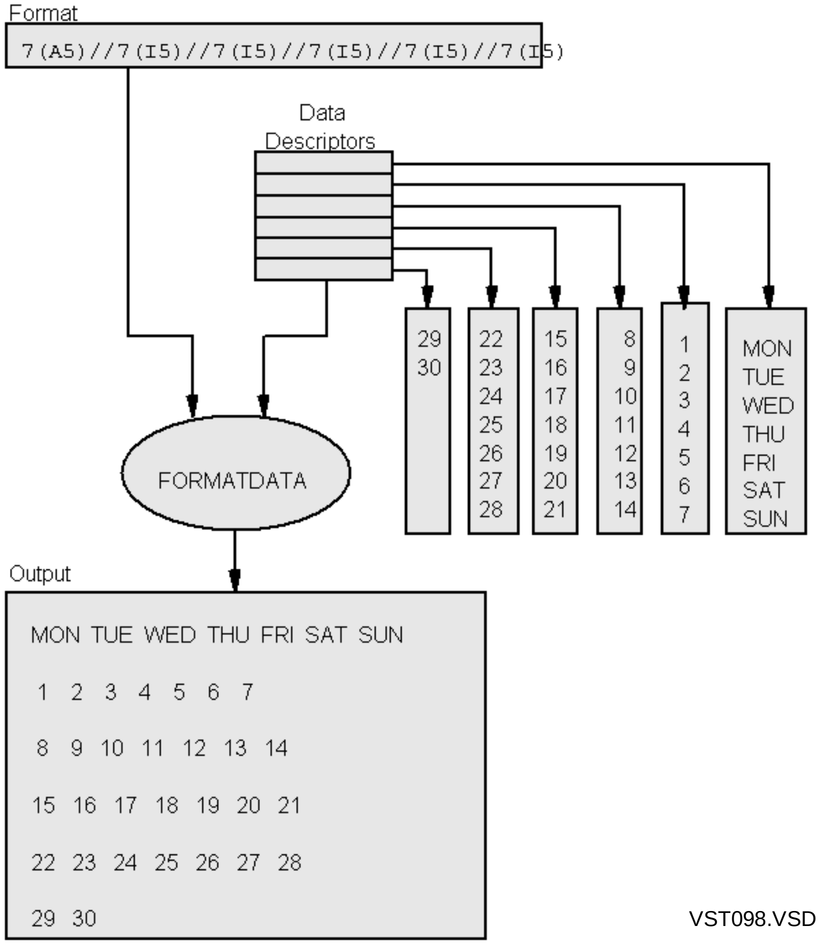
The following code fragment expands the previous example by inserting two new-buffer characters between the edit descriptors that correspond to the day of the week and the edit descriptors that correspond to the date. The code fragment is expanded to use 11 buffers, where one buffer contains the data for one line of a printed calendar.

```
!Set up the edit descriptors and convert to internal form:
EFORMAT ':=' ["7(A5)//7(I5)//7(I5)//7(I5)//7(I5)//7(I5)"];
SCALES := 0;
CONVERSION := 1;
ERROR := FORMATCONVERT(IFORMAT,IFORMATLEN,EFORMAT,
                       EFORMATLEN,SCALES,SCALE^COUNT,
                       CONVERSION);
IF ERROR <= 0 THEN ...

!Set up arrays for month, year, and date values:
DAYS       ':=' "   MON  TUE  WED  THU  FRI  SAT  SUN"
INT^ARRAY1 ':=' [1,2,3,4,5,6,7];
INT^ARRAY2 ':=' [8,9,10,11,12,13,14];
INT^ARRAY3 ':=' [15,16,17,18,19,20,21];
INT^ARRAY4 ':=' [22,23,24,25,26,27,28];
INT^ARRAY5 ':=' [29,30];

!Set up list elements that point to the above arrays:
VLIST^LEN := 6;
FLAGS := 0;
VLIST[0].ELEMENT^PTR := @DAYS;
VLIST[0].ELEMENT^SCALE := 0;
VLIST[0].ELEMENT^TYPE := 0;
VLIST[0].ELEMENT^LENGTH := 38;
VLIST[0].ELEMENT^OCCURS := 1;
VLIST[1].ELEMENT^PTR := @INT^ARRAY1;
VLIST[2].ELEMENT^PTR := @INT^ARRAY2;
VLIST[3].ELEMENT^PTR := @INT^ARRAY3;
VLIST[4].ELEMENT^PTR := @INT^ARRAY4;
I := 1;
WHILE I < VLIST^LEN DO
BEGIN
   VLIST[I].ELEMENT^SCALE := 0;
   VLIST[I].ELEMENT^TYPE := 2;
   VLIST[I].ELEMENT^LENGTH := 2;
   VLIST[I].ELEMENT^OCCURS := 7;
   I := I + 1;
END;
```

```
VLIST[5].ELEMENT^PTR := @INT^ARRAY5;
VLIST[5].ELEMENT^OCCURS := 2;

!Format the data:
ERROR := FORMATDATA(BUFFERS,BUFFER^LENGTH,NUM^BUFFERS,
                    BUFFER^ELEMENTS,WFORMAT,VLIST,VLIST^LEN,
                    FLAGS);
IF ERROR <> 0 THEN ...
```

Figure 19-5 shows how the code fragment presented above works.

Figure 19-5. Buffer Control


Format
7(A5)//7(I5)//7(I5)//7(I5)//7(I5)//7(I5)
Data Descriptors
29 30
22 23 24 25 26 27 28
15 16 17 18 19 20 21
8 9 10 11 12 13 14
1 2 3 4 5 6 7
MON TUE WED THU FRI SAT SUN
FORMATDATA
Output
MON TUE WED THU FRI SAT SUN
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

VST098.VSD

Formatting Literals

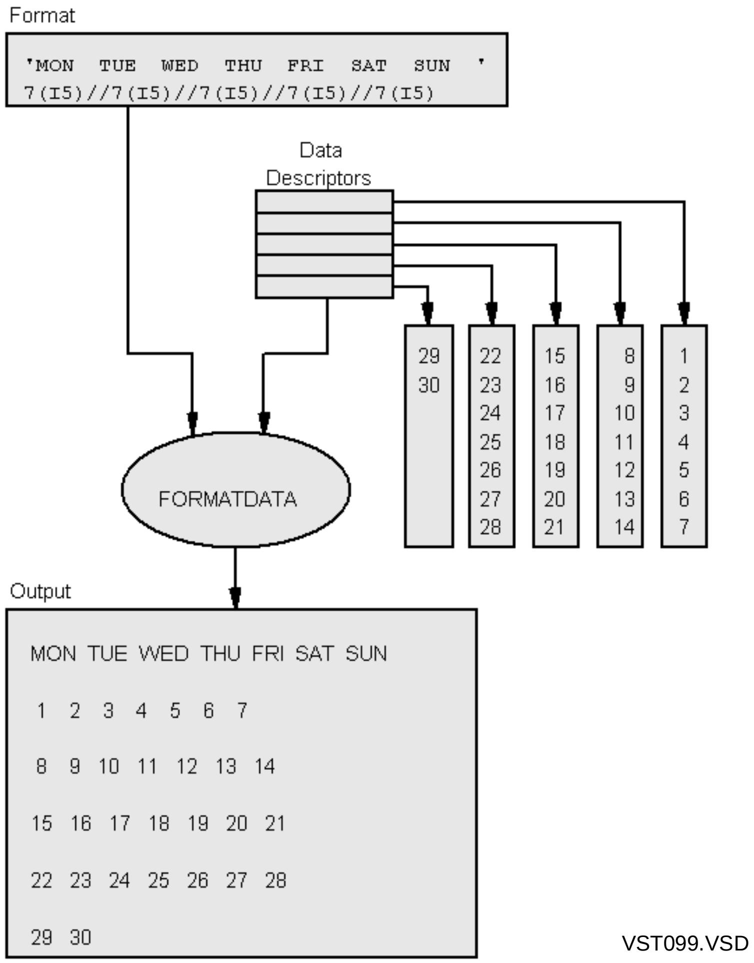
You can include literals in your edit-descriptor string by enclosing each literal in single quotation marks. The FORMATDATA[X] procedure copies these literals directly to the output buffers without accessing a data descriptor.

The following example produces the same output as the previous example. However, because the days of the week are constant values whatever the month, these values can be expressed as literals. Note that now one less data descriptor is needed.

```
!Set up the edit descriptors and convert to internal form:
EFORMAT ':=' ["'   SUN  MON  TUE  WED  THU  FRI  SAT'//,",
              "7(I5)//7(I5)//7(I5)//7(I5)//7(I5)"];
SCALES := 0;
CONVERSION := 1;
ERROR := FORMATCONVERT(IFORMAT,IFORMATLEN,EFORMAT,
                       EFORMATLEN, SCALES,SCALE^COUNT,
                       CONVERSION);
IF ERROR <= 0 THEN ...

!Set up arrays for date of the month values:
INT^ARRAY1 ':=' [1,2,3,4,5,6,7];
INT^ARRAY2 ':=' [8,9,10,11,12,13,14];
INT^ARRAY3 ':=' [15,16,17,18,19,20,21];
INT^ARRAY4 ':=' [22,23,24,25,26,27,28];
INT^ARRAY5 ':=' [29,30];

!Set up list elements that point to the above arrays:
VLIST^LEN := 5;
FLAGS := 0;
VLIST[0].ELEMENT^PTR := @INT^ARRAY1;
VLIST[1].ELEMENT^PTR := @INT^ARRAY2;
VLIST[2].ELEMENT^PTR := @INT^ARRAY3;
VLIST[3].ELEMENT^PTR := @INT^ARRAY4;
I := 0;
WHILE I < VLIST^LEN DO
BEGIN
   VLIST[I].ELEMENT^SCALE := 0;
   VLIST[I].ELEMENT^TYPE := 2;
   VLIST[I].ELEMENT^LENGTH := 2;
   VLIST[I].ELEMENT^OCCURS := 7;
   I := I + 1;
END;
VLIST[4].ELEMENT^PTR := @INT^ARRAY5;
VLIST[4].ELEMENT^OCCURS := 2;

!Format the data:
ERROR := FORMATDATA(BUFFERS,BUFFER^LENGTH,NUM^BUFFERS,
                    BUFFER^ELEMENTS,WFORMAT,
                    VLIST,VLIST^LEN,
                    FLAGS);
IF ERROR <> 0 THEN ...
```

Figure 19-6 shows the effect of the above code fragment.

Figure 19-6. Formatting Literals


Format
'MON TUE WED THU FRI SAT SUN '
7(I5)//7(I5)//7(I5)//7(I5)//7(I5)
Data Descriptors
29 30
22 23 24 25 26 27 28
15 16 17 18 19 20 21
8 9 10 11 12 13 14
1 2 3 4 5 6 7
FORMATDATA
Output
MON TUE WED THU FRI SAT SUN
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
VST099.VSD

Tabulating Data

You can tabulate data by including tabulation edit descriptors in the edit-descriptor string. Any of the following are valid forms of tabulation descriptor:

T*n*	Transmission of a character to or from a buffer is to occur at the *n*th character position in the buffer. The first character in the buffer is numbered 1.
TL*n*	Transmission of the next character to or from a buffer is to occur at *n* character positions to the left of the current position.
TR*n*	Transmission of the next character to or from a buffer is to occur at *n* character positions to the right of the current position.
*n*X	This edit descriptor is identical to TR*n*.

Each of these edit descriptors alters the current position but has no other effect.

The following example enhances the previous example by adding a line at the beginning of the output to include the name of the month in the middle of the line and the year number at the beginning and end of the line. The example uses tabulation descriptors to accomplish this. Note that the example also uses tabulation descriptors to locate each day of the week in the third buffer.

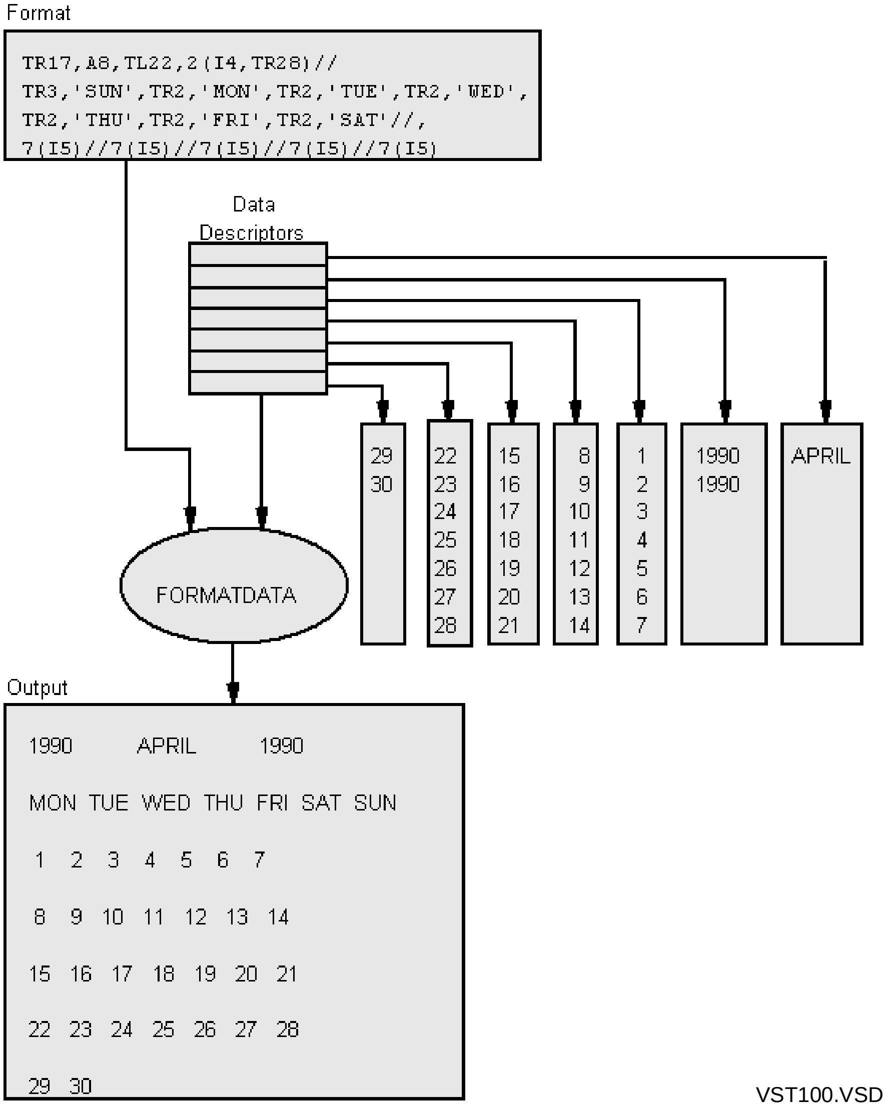
```
!Set up the edit descriptors and convert to internal form:
EFORMAT ':=' ["TR17,A8,TL22,2(I4,TR28),//,",
              "TR3,'SUN',TR2,'MON',TR2,'TUE',TR2,'WED',",
              "TR2,'THU',TR2,'FRI',TR2,'SAT'//,",
              "7(I5)//7(I5)//7(I5)//7(I5)//7(I5)"];
SCALES := 0;
CONVERSION := 1;
ERROR := FORMATCONVERT(IFORMAT,
                       IFORMATLEN,
                       EFORMAT,
                       EFORMATLEN,
                       SCALES,
                       SCALE^COUNT,
                       CONVERSION);
IF ERROR <= 0 THEN ...

!Set up arrays for month, year, and date values:
MONTH ':=' "APRIL";
INT^YEAR   ':=' [1990,1990];
INT^ARRAY1 ':=' [1,2,3,4,5,6,7];
INT^ARRAY2 ':=' [8,9,10,11,12,13,14];
INT^ARRAY3 ':=' [15,16,17,18,19,20,21];
INT^ARRAY4 ':=' [22,23,24,25,26,27,28];
INT^ARRAY5 ':=' [29,30];
```

```
!Set up list elements that point to the above arrays:
VLIST^LEN := 7;
FLAGS := 0;
VLIST[0].ELEMENT^PTR := @MONTH;
VLIST[0].ELEMENT^SCALE := 0;
VLIST[0].ELEMENT^TYPE := 0;
VLIST[0].ELEMENT^LENGTH := 10;
VLIST[0].ELEMENT^OCCURS := 1;
VLIST[1].ELEMENT^PTR := @INT^YEAR;
VLIST[1].ELEMENT^SCALE := 0;
VLIST[1].ELEMENT^TYPE := 2;
VLIST[1].ELEMENT^LENGTH := 2;
VLIST[1].ELEMENT^OCCURS := 2;
VLIST[2].ELEMENT^PTR := @INT^ARRAY1;
VLIST[3].ELEMENT^PTR := @INT^ARRAY2;
VLIST[4].ELEMENT^PTR := @INT^ARRAY3;
VLIST[5].ELEMENT^PTR := @INT^ARRAY4;
I := 2;
WHILE I < VLIST^LEN DO
BEGIN
   VLIST[I].ELEMENT^SCALE := 0;
   VLIST[I].ELEMENT^TYPE := 2;
   VLIST[I].ELEMENT^LENGTH := 2;
   VLIST[I].ELEMENT^OCCURS := 7;
   I := I + 1;
END;
VLIST[6].ELEMENT^PTR := @INT^ARRAY5;
VLIST[6].ELEMENT^OCCURS := 2;

!Format the data:
ERROR := FORMATDATA(BUFFERS,
                    BUFFER^LENGTH,
                    NUM^BUFFERS,
                    BUFFER^ELEMENTS,
                    WFORMAT,
                    VLIST,
                    VLIST^LEN,
                    FLAGS);
IF ERROR <> 0 THEN ...
```

The above code fragment is shown again at the end of this subsection as a complete program including all data declarations and relevant error checking.

Figure 19-7 shows the function of the above code fragment.

Figure 19-7. Tabulating Data



Format

```
TR17,A8,TL22,2(I4,TR28)//
TR3,'SUN',TR2,'MON',TR2,'TUE',TR2,'WED',
TR2,'THU',TR2,'FRI',TR2,'SAT'//,
7(I5)//7(I5)//7(I5)//7(I5)//7(I5)
```

Output

```
1990       APRIL       1990

MON  TUE  WED  THU  FRI  SAT  SUN

 1    2    3    4    5    6    7

 8    9   10   11   12   13   14

15   16   17   18   19   20   21

22   23   24   25   26   27   28

29   30
```

Data Descriptors
FORMATDATA
29 30
22 23 24 25 26 27 28
15 16 17 18 19 20 21
8 9 10 11 12 13 14
1 2 3 4 5 6 7
1990 1990
APRIL
VST100.VSD

Applying a Scale Factor

You can apply a scale factor to move the position of the decimal point in a fixed-point or floating-point number. Once you set a scale factor, it remains in effect until you change it.

The scale factor descriptor has the format P*n*, where *n* is the number of places by which the implied decimal point moves.

This edit descriptor affects all subsequent D, E, F, and G edit descriptors. Compare the following two sets of examples. The first set shows the results of formatting a fixed-point number and a floating-point number without a scale factor; the second set shows the same numbers formatted with a scale factor of 3:

Without a scale factor:

Format: F10.4,E12.3

Data values: 123.4567,123.4567

Output: 123.4567   0.123E+03

With a scale factor of 3:

Format: P3,F10.4,E12.3

Data values: 123.4567,123.4567

Output: 123456.7   0.123E+06

Applying Optional Plus Control

You can control whether the formatter precedes positive numbers transmitted into the output buffer with a plus sign. By default, positive numbers on output are not preceded by a plus sign.

You can use the SP edit descriptor to cause FORMATDATA[X] to put the plus signs into the output buffer. Once you specify plus signs, every positive number is displayed with a plus sign until you turn off the plus by using the S or SS edit descriptor.

The following example shows the use of the edit descriptors used in plus control, assuming that plus control is initially turned off:

Format: I4,SP,I4,I4,SS,I4

Data values: 34,45,56,67

Output: 34 +45 +56 67

Sample Program: Formatting Output

The following sample program is a complete program for printing the calendar page as illustrated in Figure 19-7. This example shows all data declarations and includes error checking for the FORMATCONVERT and FORMATDATA procedures.

The last lines of the example print out the contents of the buffers on the home terminal.

```
?INSPECT,SYMBOLS, NOLIST
?SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

!Global literals and variables:

LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;

INT   ERROR;
INT   TERM^NUM;

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FORMATCONVERT,FORMATDATA,
?               PROCESS_GETINFO_,FILE_OPEN_,WRITEX,
?               INITIALIZER,DNUMOUT,DEBUG,PROCESS_STOP_)
?LIST

!------------------------------------------------------------
! Here are some DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE =   @S^PTR := @ERROR^BUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) =   S^PTR ':=' S -> @S^PTR #;

!  Put and integer into the line:

   DEFINE PUT^INT(N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE PRINT^LINE =
         CALL WRITE^LINE(ERROR^BUFFER,
                         @S^PTR '-' @ERROR^BUFFER)   #;
```

```
!------------------------------------------------------------
! Procedure to write a line on the terminal.
!------------------------------------------------------------

PROC WRITE^LINE (BUF, LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure to display formatted data on the terminal.
!------------------------------------------------------------

PROC DISPLAY^MONTH;
BEGIN

!  Literals and variables used by FORMATCONVERT:

   LITERAL EFORMATMAXLEN = 256;       !max length of external
                                      ! edit descriptors
   STRING .EFORMAT[0:EFORMATMAXLEN - 1]; !array for external
                                         ! edit descriptors
   INT     EFORMATLEN;                !length of external
                                      ! edit descriptor
                                      ! string
   LITERAL IFORMATLEN = 512;          !max length of internal
                                      ! edit descriptors
   INT    .WFORMAT[0:IFORMATLEN/2];   !word array for edit
                                      ! descriptors passed to
                                      ! FORMATDATA
   STRING .IFORMAT := @WFORMAT '<<' 1;!string array for
                                      ! edit descriptors
                                      ! created by
                                      ! FORMATCONVERT
   INT     SCALES,                    !scale factor for
                                      ! decimal point
           SCALE^COUNT,               !number of scales
                                      ! arrays
           CONVERSION;                !type of conversion
```

```
!  Literals and variables used by FORMATDATA:

   LITERAL BUFFER^LENGTH = 80;        !length of one output
                                      ! buffer
   STRUCT  BUF^REF(*);                !data structure for an
   BEGIN                              ! output buffer
      STRING BYTES[0:BUFFER^LENGTH - 1];
   END;

   LITERAL NUM^BUFFERS = 13;          !max number of output
                                      ! buffers
   STRUCT .BUFFERS(BUF^REF) [0:NUM^BUFFERS - 1]; !Data
                                      ! structures for output
                                      ! buffers
   INT    .BUFFER^ELEMENTS [0:NUM^BUFFERS - 1]; !array to
                                      ! contain sizes of
                                      ! each output buffer

   STRUCT  VLE^REF(*);                !data structure for a
   BEGIN                              ! list element and
      INT    ELEMENT^PTR;             ! element attributes
      STRING ELEMENT^SCALE,ELEMENT^TYPE;
      INT    ELEMENT^LENGTH,ELEMENT^OCCURS;
   END;
   STRUCT   .VLIST(VLE^REF) [0:6];    !arrays for each list
                                      ! element
   INT       VLIST^LEN;               !number of list
                                      ! elements
   INT(32)  .INT^ARRAY1[0:6];         !array for list
                                      ! elements
   INT(32)  .INT^ARRAY2[0:6];         !array for second list
   INT(32)  .INT^ARRAY3[0:6];         !array for third list
   INT(32)  .INT^ARRAY4[0:6];         !array for fourth list
   INT(32)  .INT^ARRAY5[0:1];         !array for fifth list

   INT(32)  .INT^YEAR[0:1];           !variable for year
                                      ! number
   STRING    MONTH[0:9];              !month name
   INT       FLAGS;                   !flag values for
                                      ! FORMATDATA

!  Other variables:

   INT      I;                        !count
   STRING  .S^PTR;                    !string pointer
   STRING  .ERROR^BUFFER[0:40];       !buffer for error
                                      ! messages
!  Set up the edit descriptors and convert to internal form:

   EFORMAT ':=' ["TR17,A8,TL22,2(I4,TR28),//,",
                 "TR3,'SUN',TR2,'MON',TR2,'TUE',TR2,'WED',",
                 "TR2,'THU',TR2,'FRI',TR2,'SAT'//",
                  "7(I5)//7(I5)//7(I5)//7(I5)//7(I5)"]
                  -> @S^PTR;
   SCALES := 0;
   CONVERSION := 1;
   EFORMATLEN := @S^PTR '-' @EFORMAT;
   ERROR := FORMATCONVERT(IFORMAT,IFORMATLEN,EFORMAT,
                          EFORMATLEN, SCALES, SCALE^COUNT,
                          CONVERSION);
   IF ERROR <= 0 THEN
   BEGIN
      START^LINE;
      IF ERROR = 0 THEN
         PUT^STR("Internal Format Buffer Too Short")

      ELSE
      BEGIN
         PUT^STR("Format Error at Byte Position ");
         ERROR := -ERROR;
         PUT^INT(ERROR);
      END;
      PRINT^LINE;
      CALL PROCESS_STOP_;
   END;

!  Set up arrays for month, year, and date values:

   MONTH ':=' "APRIL";
   INT^YEAR   ':=' [1990D,1990D];
   INT^ARRAY1 ':=' [1D,2D,3D,4D,5D,6D,7D];
   INT^ARRAY2 ':=' [8D,9D,10D,11D,12D,13D,14D];
   INT^ARRAY3 ':=' [15D,16D,17D,18D,19D,20D,21D];
   INT^ARRAY4 ':=' [22D,23D,24D,25D,26D,27D,28D];
   INT^ARRAY5 ':=' [29D,30D];
```

```
!  Set up list elements that point to the above arrays:

   VLIST^LEN := 7;
   FLAGS := 0;
   VLIST[0].ELEMENT^PTR := @MONTH;
   VLIST[0].ELEMENT^SCALE := 0;
   VLIST[0].ELEMENT^TYPE := 0;
   VLIST[0].ELEMENT^LENGTH := 10;
   VLIST[0].ELEMENT^OCCURS := 1;
   VLIST[1].ELEMENT^PTR := @INT^YEAR;
   VLIST[1].ELEMENT^SCALE := 0;
   VLIST[1].ELEMENT^TYPE := 4;
   VLIST[1].ELEMENT^LENGTH := 4;
   VLIST[1].ELEMENT^OCCURS := 2;
   VLIST[2].ELEMENT^PTR := @INT^ARRAY1;
   VLIST[3].ELEMENT^PTR := @INT^ARRAY2;
   VLIST[4].ELEMENT^PTR := @INT^ARRAY3;
   VLIST[5].ELEMENT^PTR := @INT^ARRAY4;

   I := 2;
   WHILE I < VLIST^LEN DO
   BEGIN
      VLIST[I].ELEMENT^SCALE := 0;
      VLIST[I].ELEMENT^TYPE := 4;
      VLIST[I].ELEMENT^LENGTH := 4;
      VLIST[I].ELEMENT^OCCURS := 7;
      I := I + 1;
   END;
   VLIST[6].ELEMENT^PTR := @INT^ARRAY5;
   VLIST[6].ELEMENT^OCCURS := 2;

!  Format the data:

   ERROR := FORMATDATA(BUFFERS,        !an array of output
                                       ! buffers
                       BUFFER^LENGTH,  !length of one output
                                       ! buffer
                       NUM^BUFFERS,    !number of output
                                       ! buffers
                       BUFFER^ELEMENTS,!array for size of
                                       ! each output buffer
                       WFORMAT,        !internal format
                                       ! definition
                       VLIST,          !array of list
                                       ! elements
                       VLIST^LEN,      !number of list
                                       ! elements
                       FLAGS);         !flags for procedure
```

```
!  Check for errors:

   IF ERROR <> 0 THEN
   BEGIN
      START^LINE;
      CASE ERROR OF
      BEGIN
         267 -> PUT^STR("Buffer Overflow");
         268 -> PUT^STR("No Buffer");
         270 -> PUT^STR("Format Loopback");
         271 -> PUT^STR("EDIT Item Mismatch");
         272 -> PUT^STR("Illegal Input Character");
         273 -> PUT^STR("Bad Format");
         274 -> PUT^STR("Numeric Overflow");
         OTHERWISE -> PUT^STR("Unexpected Error");
      END;
      PRINT^LINE;
      CALL PROCESS_STOP_;
   END;

!  Print the contents of the buffers on the terminal:

   I := 0;
   WHILE I < NUM^BUFFERS AND BUFFER^ELEMENTS[I] >= 0 DO
   BEGIN
      CALL WRITEX(TERM^NUM,BUFFERS[I],BUFFER^ELEMENTS[I]);
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Main procedure performs initialization.
!------------------------------------------------------------

PROC CALENDAR MAIN;
BEGIN
   STRING   .TERM^NAME[0:MAXFLEN - 1];
   INT       TERMLEN;

!  Read the Startup message:

   CALL INITIALIZER;

!  Open the home terminal:

   ERROR := PROCESS_GETINFO_(!process^handle!,
                             !file^name:maxlen!,
                             !file^name^len!,
                             !priority!,
                             !moms^processhandle!,
                             TERM^NAME:MAXFLEN,
                             TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Call the DISPLAY^MONTH procedure:

   CALL DISPLAY^MONTH;

END;
```

List-Directed Formatting

List-directed formatting provides the data-conversion capabilities of the formatter without needing a specified format. The FORMATDATA[X] procedure determines the details of the data conversion based on the data types specified in the data descriptors.

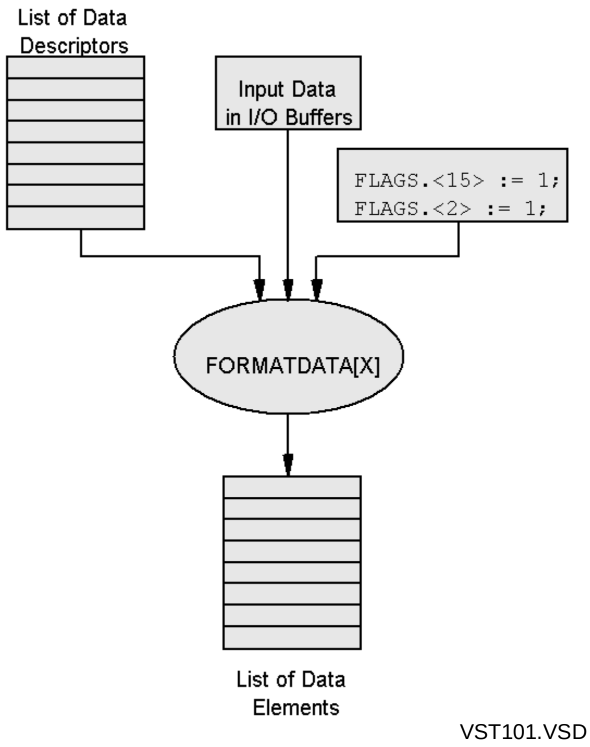
List-directed formatting can be applied to input or output as follows:

- Applied to input, the rules for list-directed formatting permit free-format input of data values rather than require fixed fields as you would need for format-directed formatting. The FORMATDATA[X] procedure converts the input data according to the data types specified in the data descriptor list and stores the converted values as indicated by the data descriptor.
- Applied to output, list-directed formatting has fewer advantages because without a specified format, the output is not necessarily in a conveniently readable form.

Formatting List-Directed Input

Figure 19-8 shows how list-directed formatting works for input.

Figure 19-8. List-Directed Formatting



To specify list-directed formatting, you need to set bit 2 of the *`flags`* parameter to 1. To specify input, you set bit 15 of the flags parameter to 1.

Here, the FORMATDATA[X] procedure takes data values from the input buffers and matches them with data descriptors from the data descriptor list. The first data value is matched with the first data descriptor, and so on. The format of the data descriptor is the same as that for format-directed formatting and is shown in Figure 19-3.

Data values provided in the input buffers are usually separated by either commas or any number of spaces. You can also separate data values using the slash character (/), which causes all subsequent values to be ignored and treated as null values.

Note. Be sure to terminate the last value in your input buffer with a value-separation character. Failure to do so causes the FORMATDATA[X] procedure to read beyond your intended input and either successfully read the wrong data or return error 272 (illegal input character).

In addition to the value-separation characters described earlier, you also need to be aware of the following rules and special values:

- Data to be saved as character strings must be enclosed in single quotation marks in the input buffer; otherwise, the FORMATDATA[X] procedure will return error 272 (illegal input character). Any other special characters, such as spaces, commas, slashes, and asterisks, can appear in the string. For example:

  ```
  'This is a string'
  ```

- You can specify repeated data items in the input buffers using the asterisk (*) character. For example, to repeat the number 57 ten times, you would put the following in the input buffer:

  ```
  10*57
  ```

  To repeat a string of characters:

  ```
  5*'TANDEM'
  ```

- You can specify a null value by placing two consecutive commas in the input buffer, optionally separated by spaces:

  ```
  , ,
  ```

  You can also specify a series of null values with a special use of the * operator. The following example specifies seven consecutive null values:

  ```
  7*
  ```

  A null value has no effect on the corresponding data item.

FORMATDATA[X] converts the data value as specified by the data type and places the converted value in the variable indicated by the data pointer in the data descriptor.

Sample Program: Formatting List-Directed Input

The following sample program formats input using a list of data descriptors. The program prompts the user for input, converts the input to internal format, and then stores the converted form.

Specifically, the code prompts the user twice: once to enter a date and once to enter a name. The user responds to the first prompt by entering the month, day of the month, and year. The user can enter this information in free format, separating each value from the next either by a comma or by one or more spaces. Note that the value representing the month is a character string and must therefore be enclosed in single quotation marks. For example:

```
Enter 'month' date year:
'May'   3   1990
```

The program puts the input values into the first buffer. Note that because the program fills the buffer with blanks before reading from the terminal, there is no need for the user to type a value-separating character after typing the year number.

The user responds to the second prompt by typing a name. The name is a character string and must be enclosed in single quotation marks:

```
Enter your name (up to 20 characters):
'Tom Sawyer'
```

The program puts the name into the second of the input buffers. Once again, the program fills the buffer with blanks before reading from the terminal, eliminating the need to type a value-termination character after the name.

The program calls the FORMATDATA procedure to convert the data in the input buffers. FORMATDATA reads the buffers left-to-right, starting with the first buffer.

FORMATDATA uses the first data descriptor (VLIST[0]) in the list of data descriptors to format the first value: the month. Note that the data type is specified by the data descriptor as type 0 (character data). If the value in the input buffer is enclosed in quotation marks, then FORMATDATA places that value into a 10-element string array pointed to by @MONTH. If the input value is not in single quotation marks, then FORMATDATA returns error 272 (illegal input character).

Similarly, FORMATDATA reads the second value in the input buffer and processes it using data descriptor VLIST[1], and so on.

When handling potential errors, this program prompts you to enter your data again if the error is of a type that is caused by entering incorrect data. For nonrecoverable errors, the program prints a diagnostic message and exits.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!Global literals and variables:

LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !max file-name
                                         ! length
LITERAL BUFSIZE = 512;     !size of I/O buffer
INT     TERM^NUM;          !file number for terminal
INT     ERROR;             !returned by system procedures
STRING .S^PTR;             !string pointer
STRING .SBUFFER[0:511];    !buffer for terminal I/O

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FORMATDATA,PROCESS_GETINFO_,
?               PROCESS_STOP_,WRITEX,WRITEREADX,FILE_OPEN_,
?               INITIALIZER)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =   @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE  PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Print the line:

   DEFINE PRINT^LINE =
       CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE  PRINT^BLANK =
           CALL WRITE^LINE(SBUFFER, 0)  #;


!------------------------------------------------------------
! Procedure to print text on the terminal.
!------------------------------------------------------------

PROC WRITE^LINE (BUF, LEN);
STRING  .BUF;
INT      LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure to format list-directed input.
!------------------------------------------------------------

PROC FORMAT^INPUT;

BEGIN

!  Literals and variables used by FORMATDATA:

   LITERAL BUFFER^LENGTH = 80;    !length of one input buffer
   STRUCT  BUF^REF(*);            !structure definition for
   BEGIN                          ! an input buffer
      STRING BYTES[0:BUFFER^LENGTH - 1];
   END;
   LITERAL NUM^BUFFERS = 2;       !max number of input
                                  ! buffers
   STRUCT .BUFFERS(BUF^REF) [0:NUM^BUFFERS - 1];!data
   INT    .BUFFER^ELEMENTS [0:NUM^BUFFERS - 1]; ! structures
                                  ! for input buffers

   STRUCT VLE^REF(*);             !data structure definition
   BEGIN                          ! for a list element and
      INT  ELEMENT^PTR;           ! its attributes
      STRING ELEMENT^SCALE,ELEMENT^TYPE;
      INT    ELEMENT^LENGTH,ELEMENT^OCCURS;
   END;
   STRUCT .VLIST(VLE^REF) [0:3];  !arrays for each data
                                  ! descriptor
   INT     VLIST^LEN;             !number of data descriptors
   INT     WFORMAT;               !dummy internal format
   INT     FLAGS;                 !flag values for FORMATDATA

!  The list elements:

   STRING .MONTH[0:9],            !month name
          .NAME[0:19];            !user name
   INT     DATE,                  !date of month
           YEAR;                  !year number

!  Other variables:

   INT     BYTES^READ;            !used by I/O procedures
   INT     I;                     !count
```

```
!  Initialize variables for month, date, year, and name:

   MONTH ':=' "          ";
   DATE := 0;
   YEAR := 0000;
   NAME ':=' [10 * ["  "]];

!  Set up data descriptors that point to the above
!  variables:

   VLIST[0].ELEMENT^PTR    := @MONTH;
   VLIST[0].ELEMENT^SCALE  := 0;
   VLIST[0].ELEMENT^TYPE   := 0;
   VLIST[0].ELEMENT^LENGTH := 10;
   VLIST[0].ELEMENT^OCCURS := 1;

   VLIST[1].ELEMENT^PTR    := @DATE;
   VLIST[1].ELEMENT^SCALE  := 0;
   VLIST[1].ELEMENT^TYPE   := 2;
   VLIST[1].ELEMENT^LENGTH := 2;
   VLIST[1].ELEMENT^OCCURS := 1;

   VLIST[2].ELEMENT^PTR    := @YEAR;
   VLIST[2].ELEMENT^SCALE  := 0;
   VLIST[2].ELEMENT^TYPE   := 2;
   VLIST[2].ELEMENT^LENGTH := 2;
   VLIST[2].ELEMENT^OCCURS := 1;

   VLIST[3].ELEMENT^PTR    := @NAME;
   VLIST[3].ELEMENT^SCALE  := 0;
   VLIST[3].ELEMENT^TYPE   := 0;
   VLIST[3].ELEMENT^LENGTH := 20;
   VLIST[3].ELEMENT^OCCURS := 1;

!  Specify number of data descriptors:

   VLIST^LEN := 4;

!  Set flags for list-directed formatting and for input:

   FLAGS.<2> := 1;
   FLAGS.<15> := 1;
```

```
!  Repeat formatting loop if erroneous input:

PROMPT^AGAIN:

!  Blank the buffers:

   I := 0;
   WHILE I < NUM^BUFFERS DO
   BEGIN
      BUFFERS[I] ':=' [40 * ["  "]];
      I := I + 1;
   END;

!  Prompt for input and copy into buffers:

   SBUFFER ':=' "Enter 'month' date year:            "
                -> @S^PTR;
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL PROCESS_STOP_;
   BUFFERS[0] ':=' SBUFFER FOR BYTES^READ BYTES;

   SBUFFER ':=' "Enter 'name' (up to 20 characters): "
                -> @S^PTR;
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL PROCESS_STOP_;
   BUFFERS[1] ':=' SBUFFER FOR BYTES^READ BYTES;

!  Format the data:

   ERROR := FORMATDATA(BUFFERS,        !an array of input
                                       ! buffers
                       BUFFER^LENGTH,  !length of one input
                                       ! buffer
                       NUM^BUFFERS,    !number of input
                                       ! buffers
                       BUFFER^ELEMENTS,!unused
                       WFORMAT,        !internal format
                                       ! definition (= 0)
                       VLIST,          !array of data
                                       ! descriptors
                       VLIST^LEN,      !number of data
                                       ! descriptors
                       FLAGS);         !flags for
                                       ! procedure
!  Check for errors. If invalid input then retry, otherwise
!  stop:

   IF ERROR <> 0 THEN
   BEGIN
      START^LINE;
      CASE ERROR OF
      BEGIN
         267 -> PUT^STR("Buffer Overflow");
         272 -> PUT^STR("Illegal Input Character");
         OTHERWISE -> BEGIN
            CASE ERROR OF
            BEGIN
               268 -> PUT^STR("No Buffer");
               270 -> PUT^STR("Format Loopback");
               271 -> PUT^STR("EDIT Item Mismatch");
               273 -> PUT^STR("Bad Format");
               274 -> PUT^STR("Numeric Overflow");
               OTHERWISE -> PUT^STR("Unexpected Error" &
                                     "Number");
            END;
            PRINT^LINE;
            CALL PROCESS_STOP_;
         END;
      END;
      PRINT^LINE;
      START^LINE;
      PUT^STR("Reenter Your Data ");
      PRINT^LINE;
      PRINT^BLANK;
      GOTO PROMPT^AGAIN;
   END;
END;
```

```
!------------------------------------------------------------
! Main procedure provides initialization
!------------------------------------------------------------

PROC INIT MAIN;
BEGIN

   STRING   .TERM^NAME[0:MAXFLEN - 1];
   INT       TERMLEN;

!  Read the Startup message:

   CALL INITIALIZER;

!  Open the home terminal:

   ERROR := PROCESS_GETINFO_(!process^handle!,
                             !file^name:maxlen!,
                             !file^name^len!,
                             !priority!,
                             !moms^processhandle!,
                             TERM^NAME:MAXFLEN,
                             TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Call the FORMAT^INPUT procedure:

   CALL FORMAT^INPUT;

END;
```

Formatting List-Directed Output

List-directed output works like list-directed input in reverse. Here, the FORMATDATA[X] procedure takes data from variables addressed by a list of data descriptors and writes them to an output buffer in a format that depends on the data type specified in the data descriptor.

For example, if the data type is character, then the stored information is interpreted as ASCII code. If the data type is 16-bit integer, then each stored word is treated as an integer value, converted to ASCII code, and written to the output buffer.

You specify list-directed output by setting bits in the *`flags`* parameter supplied to the FORMATDATA[X] procedure. Set bit 15 to 0 to specify output. Set bit 2 to 1 to specify list-directed formatting.

Manipulating Character Strings

Without using the formatter, there are several operations that you can perform on character strings:

- Convert a string of ASCII numeric characters into a binary number (NUMIN and DNUMIN procedures) or convert a binary number into an ASCII string for output (NUMOUT and DNUMOUT procedures). See Converting Between Strings and Integers later in this subsection.
- Change lowercase alphabetic characters into uppercase or change uppercase alphabetic characters into lowercase (SHIFTSTRING procedure). Case Shifting Character Strings describes how to do this.
- Edit a string (FIXSTRING procedure). See Editing a Character String.
- Sort characters in memory (HEAPSORT[X_] procedure). See Sorting Characters.

Converting Between Strings and Integers

Numeric input and output to a terminal is done using standard 7-bit ASCII codes. Internally, numeric representation takes the form of binary numbers. You therefore need to convert from ASCII to binary numeric representation on input and from binary to ASCII representation on output.

One way of converting between ASCII and binary numeric representation is to use the formatter as described in the previous subsection. The formatter can perform this conversion for any numeric type. For single-length and double-length integers, however, you can use the NUMIN, DNUMIN, NUMOUT, and DNUMOUT procedures. The following paragraphs describe how.

Converting a Numeric ASCII String Into a Binary Number

To convert a numeric ASCII string into a binary number, you use either the NUMIN or DNUMIN procedure. For a 16-bit result, you use the NUMIN procedure. For a 32-bit result, use the DNUMIN procedure.

You must supply the ASCII number that you want to convert, along with the numeric base of the ASCII number. The numeric base must be in the range 2 through 10 for NUMIN or 2 through 36 for DNUMIN. NUMIN or DNUMIN recognizes the end of the numeric string by the first nonnumeric or zero character in the input buffer.

The NUMIN procedure returns the signed 16-bit result and a status indication showing whether the conversion was successful. DNUMIN provides the same information as NUMIN except that the result is 32 bits. Both procedures also return the address of the first character after the input string. You can use this value to check that the procedure converted the expected number of characters.

The following example reads some ASCII input from a terminal. The input is expected to be numeric data so the DNUMIN procedure is used to convert the number from ASCII representation into a binary number.

```
BASE := 10;

!Read from the terminal into the input buffer:
CALL READX(TERMNUM,SBUFFER,BUFSIZE,COUNT^READ);
IF <> THEN ...                     !file-system error

!Set the next byte in the buffer to zero to make sure that
!DNUMIN recognizes the end of the numeric string
SBUFFER[COUNT^READ] := 0;
@NEXT^ADDR := DNUMIN(SBUFFER,       !numeric ASCII code
                     SIGNED^RESULT, !32-bit result
                     BASE,          !numeric base of input
                     STATUS);       !status of conversion

!Check that the value of STATUS is zero and that DNUMIN
!converted the expected number of characters:
IF STATUS OR @NEXT^ADDR <> @SBUFFER[COUNT^READ]
         THEN ...                  !invalid number
```

Converting a Binary Number Into an ASCII String

To convert a binary number into an ASCII string, you use either the NUMOUT or DNUMOUT procedure. For a 16-bit integer, you use the NUMOUT procedure. For a 32-bit integer, you use the DNUMOUT procedure.

To use the NUMOUT procedure, you must supply the 16-bit binary integer that you want to convert, along with the numeric base you require for the ASCII number and the maximum number of characters you permit in the output. The numeric base must be in the range 2 through 10. The NUMOUT procedure returns the ASCII result. An example follows:

```
BASE := 10;
WIDTH := 4;
CALL NUMOUT(ASCII^RESULT,        !output string
            UNSIGNED^INTEGER,    !binary input
            BASE,                !numeric base of output
            WIDTH);              !maximum number of
                                 !characters in output
```

Case Shifting Character Strings

You should use the SHIFTSTRING procedure to perform all case-shifting operations on alphabetic characters. This procedure enables you to perform case shifting from lowercase to uppercase and from uppercase to lowercase.

The standard ASCII character set allows you to shift case by inverting the fifth bit from the right of any alphabetic character. However, not every local character set uses this mechanism for case shifting. You are therefore encouraged to use the SHIFTSTRING procedure, which is configured to work with the locally supported character set.

Upshifting a Character String

To upshift a character string, you supply the SHIFTSTRING procedure with the string to be upshifted and the length of the string. To specify upshifting, you must set bit 15 of the *casebit* parameter to 0 (the default value). The following example converts an input string to all uppercase letters:

```
STRING ':=' INPUT^BUFFER FOR 10;
STRING^LENGTH := 10;
CASE^BIT.<15> := 0;
CALL SHIFTSTRING(STRING,
                 STRING^LENGTH,
                 CASE^BIT);
```

Any nonalphabetic characters in the input string remain unchanged. Uppercase alphabetic characters in the input string also remain unchanged.

Downshifting a Character String

To downshift any uppercase alphabetic characters in a string, you should use the SHIFTSTRING procedure with bit 15 of the *casebit* parameter set to 1. For example:

```
CASE^BIT.<15> := 1;
CALL SHIFTSTRING(STRING,
                 STRING^LENGTH,
                 CASE^BIT);
```

Editing a Character String

The FIXSTRING procedure edits a string based on commands provided in a template. The FIXSTRING procedure is commonly used in an interactive process to implement a command that edits command strings. For example, the FC command uses the FIXSTRING procedure to edit any other command; see the *Guardian User's Guide* for details. Likewise, the FC command in Debug uses the FIXSTRING procedure; see the *Inspect Manual* for details.

The FIXSTRING procedure works by supplying the ASCII string you want to edit and a template containing the edit commands. The ASCII string can be any sequence of ASCII characters whose length can be limited by supplying the *max-data-len* parameter. The template contains commands for replacing, deleting, and inserting characters.

The following TAL statement shows an example of the FIXSTRING procedure:

```
CALL FIXSTRING(TEMPLATE,       !string array of edit commands
               TEMPLATE^LEN,   !length of template
               DATA,           !string to edit
               DATA^LENGTH,    !length of string
               MAX^DATA^LEN);  !maximum length of edited
                               ! string
```

Note that the data parameter contains the string to be edited on input and the edited string on output.

Using the FIXSTRING Template

You can supply any of the following three commands in the FIXSTRING template:

- R or r replaces characters in the string
- D or d deletes characters in the string
- I or i inserts characters in the string

The R or r command in the template causes all characters that follow the R or r command to replace the corresponding characters in the string. The following example shows use of the R command to replace text in the character string:

```
Before string:  fup dup filea.fileb
Template:                   R,filec
After string:   fup dup filea,filec
```

Note that the R command is implied if no other command is specified. For example, the following template achieves the same result:

```
Before string:  fup dup filea.fileb
Template:                    ,filec
After string:   fup dup filea,filec
```

The implied replace command works only if the first character of the template is not a command name (D, d, I, or i).

To delete characters in a string, the template must contain a D or d at the position where you want a character deleted. For example:

```
Before string:  fup dup filea,fileb,filec
Template:                    DDDDDD
After string:   fup dup filea,filec
```

To insert characters into a string, the template must contain an i or I character at the corresponding character position, followed by the text to be inserted:

```
Before string:  fup filea,filec
Template:          I dup
After string:   fup dup filea,filec
```

You can supply multiple commands in the same template by separating the commands with two slashes. For example:

```
Before string:   fup filea,filrv
Template:            idup //  rec
After string:    fup dup filea,filec
```

Editing Commands: An Example

The following sample program features a command interpreter with the ability to accept an FC command typed by the user. By typing “FC,” the user is given the opportunity to edit the last command entered.

The example is made up of three procedures:

- The main procedure simply calls the INITIALIZE^TERMINAL procedure to open the terminal and then calls the COMMAND^INTERPRETER procedure.
- The INITIALIZE^TERMINAL and SAVE^STARTUP^MESSAGE procedures read the Startup message, save it in a global data structure, and then open the file specified as the IN file in the Startup message.
- The COMMAND^INTERPRETER procedure prompts the user to enter a command, which can be up to eight characters long. The procedure converts any lower-case alphabetic characters to upper case, and then processes the command itself. If no such command exists, then the program displays a diagnostic message.

  If the user types the FC command, then the COMMAND^INTERPRETER procedure calls the FC procedure to edit the previous command. The FC procedure returns 1 after successfully editing the command and the COMMAND^INTERPRETER procedure executes the edited command. If FC returns 0 (without successfully editing the command), then the COMMAND^INTERPRETER procedure prompts the user for another command.

  The COMMAND^INTERPRETER procedure exits only when the user types the EXIT command.
- The FC procedure is called by the COMMAND^INTERPRETER procedure when the user types the FC command. The FC procedure displays the previous command and prompts the user to enter a template that the FIXSTRING procedure will use to edit the command. If the user types just two slash characters, then FC returns 0 to the COMMAND^INTERPRETER. Otherwise the FC procedure edits the command according to the input.

  Once FIXSTRING has edited the command, the FC procedure repeats, offering the user the chance to edit the new command. The user refuses by pressing carriage return in response to the FC prompt, which causes the FC procedure to return 1 to the COMMAND^INTERPRETER procedure.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.SYSTEM.ZSYSTAL;
?LIST

!Global literals and variables:

INT     TERM^NUM;                       !open terminal file number

STRUCT .CI^STARTUP;                     !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULT;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOLUME[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAM[0:529];
END;

LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;
LITERAL ABEND   = 1;

?NOLIST
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(WRITEX,WRITEREADX,
?   FILE_OPEN_,FIXSTRING,PROCESS_STOP_,INITIALIZER,
?   SHIFTSTRING,OLDFILENAME_TO_FILENAME_)
?LIST
```

```
!------------------------------------------------------------
! Integer procedure edits the command buffer and returns 1
! if edited command should be executed.  This procedure
! allows the user a change of mind about editing the command
! by returning 0.
!------------------------------------------------------------

INT PROC FC(COMMAND,LAST^COMMAND,NUM,SAVE^NUM);
STRING .COMMAND;
STRING .LAST^COMMAND;
INT    .NUM;
INT    .SAVE^NUM;

BEGIN
   STRING  .TEMPLATE^ARRAY[0:71]; !template used for edit
                                  ! changes
   INT      TEMPLATE^LENGTH;      !length of template
   INT      MAX^LEN := 8;         !maximum length of edited
                                  ! command
   STRING  .BUFFER[0:71];         !I/O buffer
   STRING  .S^PTR;                !pointer to end of text
                                  ! string

!  Set command prompt to "< ":

   COMMAND[-2] ':=' "< ";

!  Set NUM equal to size of previous command:

   NUM := SAVE^NUM;

!  Put previous command in command buffer:

   COMMAND ':=' LAST^COMMAND FOR NUM;

!  Edit the command each time through the loop.  The loop
!  enables the user to check the results of an edit and then
!  edit again if necessary:

   DO
   BEGIN

   !  Write the command to be edited to the terminal:

      CALL WRITEX(TERM^NUM,COMMAND[-2],NUM + 2);
      IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                    !specifier!,
                                    ABEND);
```

```
!  Set the FC prompt to " ." and read template typed by
!  user:

   TEMPLATE^ARRAY ':=' " .";
   CALL WRITEREADX(TERM^NUM,TEMPLATE^ARRAY,2,72,
                   TEMPLATE^LENGTH);

!  If WRITEREADX returns an error, or if the template
!  contains exactly two slashes, then return with no
!  changes:

   IF > OR TEMPLATE^LENGTH = 2
      AND TEMPLATE^ARRAY = "//" THEN
   BEGIN
      NUM := SAVE^NUM;
      COMMAND ':=' LAST^COMMAND FOR NUM;
      RETURN 0;
   END;

!  Otherwise call FIXTRING to alter the command according
!  to the instructions in the template:

   CALL FIXSTRING(TEMPLATE^ARRAY,TEMPLATE^LENGTH,
      COMMAND,NUM,MAX^LEN);
   IF > THEN
   BEGIN

   !  The replacement string is greater than MAX^LEN, so
   !  print a diagnostic message and return to beginning
   !  of loop:

      BUFFER ':=' "Replacement string too long "
         -> @S^PTR;
      CALL WRITEX(TERM^NUM,BUFFER,@S^PTR '-' @BUFFER);
      IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                    !specifier!,
                                    ABEND);
   END
   ELSE IF < THEN CALL PROCESS_STOP_(!process^handle!,
                                     !specifier!,
                                     ABEND);

!  Upshift all characters in the edited command in case
!  any characters were typed in lowercase:

   CALL SHIFTSTRING(COMMAND,NUM,0);
END
```

```
!  Loop until user responds to FC prompt with a carriage
!  return only:

   UNTIL NOT TEMPLATE^LENGTH;

!  Return to command interpreter to execute edited command:

   RETURN 1;

END;


!------------------------------------------------------------
! Procedure prompts the user for a command and then processes
! the command.  This procedure loops indefinitely until the
! user types the EXIT command.
!------------------------------------------------------------

PROC COMMAND^INTERPRETER;
BEGIN
   STRING .LAST^COMMAND[0:7];     !buffer for last command
   INT     NUM;                   !number of bytes
                                  ! transferred
   INT     SAVE^NUM;              !previous number of bytes
                                  ! transferred
   STRING .COMMAND[-2:7] := "< "; !command buffer
   STRING .BUFFER[0:71];          !I/O buffer
   STRING .S^PTR;                 !string pointer

   INT REPEAT := 0;               !when 0, prompt for new
                                  ! command; when 1,
                                  ! execute fixed command

!  Loop until user types "EXIT":

   WHILE 1 DO
   BEGIN

   !  If repeat not set, prompt user for a new command:

      IF NOT REPEAT THEN
      BEGIN
         COMMAND[0] := " ";
         COMMAND[1] ':=' COMMAND[0] FOR 7;
         COMMAND ':=' "< ";
         CALL WRITEREADX(TERM^NUM,COMMAND,2,9,NUM);
         IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                       !specifier!,
                                       ABEND);
      END;

   !  Upshift the command in case user typed lowercase:

      CALL SHIFTSTRING(COMMAND,NUM,0);
```

```
    !  If the command is "FC" then call the FC procedure,
    !  returning 1 if the fix is accepted or 0 if it is not.
    !  If the command is EXIT, then stop the program.
    !  If the command is any other valid command, then process
    !  the command (this program simply displays the command
    !  name).  If an illegal command, then print a diagnostic
    !  message:

       IF COMMAND = "FC"
          THEN REPEAT := FC(COMMAND,LAST^COMMAND,NUM,SAVE^NUM)

       ELSE BEGIN
         IF COMMAND = "EXIT" THEN CALL PROCESS_STOP_

          ELSE IF COMMAND = "COMMAND1" THEN
            CALL WRITEX(TERM^NUM,COMMAND,NUM)

          ELSE IF COMMAND = "COMMAND2" THEN
             CALL WRITEX(TERM^NUM,COMMAND,NUM)

          ELSE BEGIN
             BUFFER ':=' COMMAND FOR 8;
             BUFFER[8] ':=' ":  Illegal Command " -> @S^PTR;
             CALL WRITEX(TERM^NUM,BUFFER,@S^PTR '-' @BUFFER);
             IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                           !specifier!,
                                           ABEND);
          END;

       !  Reset the repeat flag:

          REPEAT := 0;
       END;

    !  If the command length is nonzero, then save it in the
    !  LAST^COMMAND array for possible editing by a subsequent
    !  FC command:

       IF NUM THEN
       BEGIN
          SAVE^NUM := NUM;
          LAST^COMMAND ':=' COMMAND FOR SAVE^NUM;
       END;
    END;
 END;
```

```
!-------------------------------------------------------------
! Procedure to save the Startup message in a global
! structure.
!-------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,LENGTH,
                          MATCH) VARIABLE;
INT .RUCB,
    .START^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;

END;


!-------------------------------------------------------------
! Procedure to open the terminal file specified in the IN
! file of the Startup message.
!-------------------------------------------------------------

PROC INITIALIZE^TERMINAL;
BEGIN

   STRING .TERM^NAME[0:MAXFLEN - 1];
   INT     TERMLEN;
   INT     ERROR;

!  Read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
               CI^STARTUP.INFILE.VOLUME,
               TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                          !specifier!,
                                          ABEND);
   ERROR :=  FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                          !specifier!,
                                          ABEND);
END;
```

```
!------------------------------------------------------------
! Main procedure initializes the IN file and then calls the
! command interpreter.
!------------------------------------------------------------

PROC INITIALIZE MAIN;
BEGIN

!  Initialize the IN file:

   CALL INITIALIZE^TERMINAL;

!  Call the command interpreter:

   CALL COMMAND^INTERPRETER;

END;
```

Sorting Characters

Use the HEAPSORT[X_] procedure to sort an array in memory. You can use the HEAPSORT procedure only to sort arrays in the user data segment; you cannot use it to sort arrays in extended memory. You can use HEAPSORTX_ to sort arrays that are either in the user data segment or in an extended data segment.

To use the HEAPSORT[X_] procedure, you must supply it with the array you want to sort, the number of elements in the array, the size of each element, and the name of the user-supplied procedure that will do the actual comparison. HEAPSORTX_ also has an optional parameter that allows you to specify an array of pointers. This array of pointers can help speed up the sort by allowing HEAPSORTX_ to sort a list of pointers instead of the data elements themselves; the pointer array is particularly useful if the sort involves a large number of elements or a large element size.

```
CALL HEAPSORTX_(ARRAY,
                NUMBER^OF^ELEMENTS,
                ELEMENT^SIZE,
                ASCENDING,          !Name of procedure to do
                                    ! comparison
                POINTER^ARRAY);
```

The following sample program sorts some strings into alphabetical order. The program is made up of three procedures:

- The main procedure provides initialization and calls the SORTING procedure.
- The SORTING procedure supplies a list of strings to the HEAPSORTX_ procedure for sorting. On return from HEAPSORTX_, the SORTING procedure displays the sorted list on the home terminal.
- The ASCENDING procedure is called by HEAPSORTX_ to compare pairs of strings. This procedure returns 1 if the first string is less than the second string or 0 if the second string is less than the first string. HEAPSORTX_ calls this procedure as many times as it needs to sort the entire list of strings.

```
?INSPECT,SYMBOLS
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL;
?LIST

!Literals:

LITERAL ELEMENT^SIZE = 6;
LITERAL MAXFLEN      = ZSYS^VAL^LEN^FILENAME;

!Global variables:

INT   TERM^NUM;
INT   ERROR;


?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(HEAPSORTX_,PROCESS_GETINFO_,
?               WRITEX,PROCESS_STOP_,INITIALIZER,FILE_OPEN_)
?LIST


!------------------------------------------------------------
! Procedure to sort two strings A and B. Returns 1 if A less
! than B, returns 0 if B less than or equal to A.
!------------------------------------------------------------

INT PROC ASCENDING(A,B);
INT .EXT A;
INT .EXT B;

BEGIN
   RETURN IF A < B FOR ELEMENT^SIZE THEN 1 ELSE 0;
END;
```

```
!-------------------------------------------------------------
! Procedure to initialize an array with string values and
! then call HEAPSORTX_ to sort them. By calling ASCENDING,
! it sorts them into ascending order.
!-------------------------------------------------------------

PROC SORTING;
BEGIN
   INT     I;                        !counting variable
   INT(32) NUMBER^OF^ELEMENTS;       !size of array to be
                                     ! sorted

   STRUCT  ARRAY^REF(*);             !structure defining an
   BEGIN                             ! array element
      STRING ELEMENT[0:11];
   END;

   STRUCT .ARRAY(ARRAY^REF)[0:9];  !array with 10 elements

!  Initialize array for sorting.  For simplicity the array
!  is initialized statically.  In practice, the array would
!  typically be read from another file or entered
!  interactively at the terminal:

   ARRAY[0] ':=' "BUSH        ";
   ARRAY[1] ':=' "REAGAN      ";
   ARRAY[2] ':=' "CARTER      ";
   ARRAY[3] ':=' "FORD        ";
   ARRAY[4] ':=' "NIXON       ";
   ARRAY[5] ':=' "JOHNSON     ";
   ARRAY[6] ':=' "KENNEDY     ";
   ARRAY[7] ':=' "EISENHOWER  ";
   ARRAY[8] ':=' "TRUEMAN     ";
   ARRAY[9] ':=' "WASHINGTON  ";

!  Sort the array:

   NUMBER^OF^ELEMENTS := 10D;
   ERROR := HEAPSORTX_(ARRAY,
                       NUMBER^OF^ELEMENTS,ELEMENT^SIZE,
                       ASCENDING);

!  Print the array in sorted order:

   I := 0;
   WHILE $DBL(I) < NUMBER^OF^ELEMENTS DO
   BEGIN
      CALL WRITEX(TERM^NUM,ARRAY[I],(ELEMENT^SIZE * 2));
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Main procedure performs initialization
!------------------------------------------------------------

PROC SORTER MAIN;
BEGIN
   STRING  .TERM^NAME[0:MAXFLEN - 1];
   INT      TERMLEN;

!  Read the Startup message:

   CALL INITIALIZER;

!  Open the home terminal:

   ERROR := PROCESS_GETINFO_(!process^handle!,
                             !file^name:maxlen!,
                             !file^name^len!,
                             !priority!,
                             !moms^processhandle!,
                             TERM^NAME:MAXFLEN,
                             TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Call the SORTING procedure to initialize the sort array:

   CALL SORTING;
END;
```

Programming With Multibyte Character Sets

The operating system provides support for national languages whose character set cannot be represented by a single-byte character set such as ASCII code. To support languages with larger character sets, such as Japanese, Korean, and Chinese, HP provides multibyte character sets.

Specifically, HP provides internal representations of the following character sets for use with terminals that support multibyte character sets:

- Tandem Kanji
- Tandem Hangul
- Tandem Chinese Big 5
- Tandem Chinese PC
- Tandem KSC5601

The operating system supports text strings that can contain codes from one of the above character sets and standard ASCII codes within the same string.

In addition to the character sets listed above, HP also provides external support for the following character sets:

- IBM Kanji
- IBM Kanji Mixed
- JEF (Fujitsu) Kanji
- JEF (Fujitsu) Kanji Mixed
- NEC Kanji
- JIS Kanji

The operating system provides procedures that convert between each of the above character sets and internal Tandem Kanji codes.

This subsection describes some of the operations that an application may need to perform with multibyte character sets:

- How to check whether multibyte support is available on your system (MBCS_CODESETS_SUPPORTED_ procedure). Checking for Multibyte Character-Set Support provides details.
- How to find out which of the multibyte character sets is the current default set (MBCS_DEFAULTCHARSET_ procedure). See Determining the Default Character Set for details.
- How to identify multibyte characters (MBCS_CHAR_ procedure). See Analyzing a Multibyte Character String for details.
- How to deal with fragments of multibyte characters that occur in the last byte of a read or write operation (MBCS_TRIMFRAGMENT_ procedure). Dealing With Fragments of Multibyte Characters provides details.
- How to handle multibyte blank characters used as word delimiters (MBCS_REPLACEBLANK_ procedure). See Handling Multibyte Blank Characters.
- How to find out the character size of a multibyte character set (MBCS_CHARSIZE_ procedure). See Determining the Character Size of a Multibyte Character Set.
- How to perform case-shift operations on multibyte characters (SHIFTSTRING and MBCS_SHIFTSTRING_ procedures). See Case Shifting With Multibyte Characters for details.

This subsection does not cover the procedures that were written primarily to support other HP subsystems, although these procedures are nonprivileged and available to all users. These procedures include:

- The MBCS_CHARSTRING_ procedure used by the SCOBOL compiler for testing a text string to see whether it contains only multibyte characters.
- The MBCS_EXTERNAL_TO_TANDEM_ procedure used by SNAX and Pathway for converting external multibyte character representation into the equivalent internal character set.
- The MBCS_TANDEM_TO_EXTERNAL_ procedure used by SNAX and Pathway to convert internal multibyte character representation into an external character set.
- The MBCS_FORMAT_ITI_BUFFER_ procedure used by SNAX to format line records for specific display devices.
- The MBCS_FORMAT_CRT_FIELD_ procedure used by SNAX to format line records for specific display devices operating in block mode.

For details on these procedures, refer to the *Guardian Procedure Calls Reference Manual*.

Checking for Multibyte Character-Set Support

Use the MBCS_CODESETS_SUPPORTED_ procedure to find out which multibyte character sets your system supports:

```
RESULT := MBCS_CODESETS_SUPPORTED_;
```

The 32-bit result indicates which internal and external multibyte character sets are supported; each supported character set is indicated by a 1 in the bit position:

bit 1	Tandem Kanji
bit 2	IBM Kanji
bit 3	IBM Kanji Mixed
bit 4	JEF (Fujitsu) Kanji
bit 5	JEF (Fujitsu) Kanji Mixed
bit 6	NEC Kanji
bit 7	JIS Kanji
bit 9	Tandem Hangul
bit 10	Chinese Big 5
bit 11	Chinese PC
bit 12	Tandem KSC5601

If the result is zero, then there is no support for multibyte character sets.

If the result indicates support for one or more external character sets, then it will also indicate support for the corresponding internal character set. For example, if IBM Kanji is supported, then Tandem Kanji is also supported. Support for an external character set also indicates that the appropriate conversion and formatting routines are available on your system.

Determining the Default Character Set

Use the MBCS_DEFAULTCHARSET_ procedure to find out which of the supported internal character sets is the default set. This value is hard coded and can therefore be changed only by reconfiguring the system using a different object module of the multibyte character-set library.

Call the MBCS_DEFAULTCHARSET_ procedure as follows:

```
RESULT := MBCS_DEFAULTCHARSET_;
```

The result indicates the default character set as follows:

0	No multibyte character set configured
1	Tandem Kanji
9	Tandem Hangul
10	Tandem Chinese Big 5
11	Tandem Chinese PC
12	Tandem KC5601

Analyzing a Multibyte Character String

Because the operating system supports mixtures of a multibyte character set and a single-byte character set, you cannot be sure without testing whether a given byte is a single-byte character, the beginning of a multibyte character, or part of a multibyte character that is not the first byte. It is important to be able to recognize the first byte of a character to make sure that string operations start on a character boundary.

To establish the identity of a character, you use the MBCS_CHAR_ procedure. To use the MBCS_CHAR_ procedure, you must supply it with a pointer to a text string and the identity of a multibyte character set. The procedure returns with an indication whether the specified byte is the start of a multibyte character or a single-byte character. The procedure also indicates whether the character belongs to the specified multibyte character set.

When you use MBCS_CHAR_ on a text string for the first time, you should set up the pointer to the first byte in the string. The first byte will always be either a single-byte character or the first byte of a multibyte character. Once the character is identified, you should advance the pointer by the length of the identified character and then test again. This way, the pointer always points to the first byte of a character.

Any value that the pointer attains is a valid starting point for any other multibyte operation.

The following example shows the intended use of the MBCS_CHAR_ procedure:

```
!Set up the pointer to address the first byte of the text
!string:
@TESTMBCSCHAR := @TEXT^STRING[0];

!Loop, checking each character, as long as you are processing
!a mixed text string:
WHILE...   !while processing mixed text string
DO
BEGIN

   !Indicate the number of bytes remaining in the text
   !string:
   CHARSIZE := number of bytes remaining in text string

   !Check whether the pointer addresses a single-byte
   !character or a multibyte character:
   IF MBCS_CHAR_(TESTMBCSCHAR,CHARSET,CHARSIZE)
   THEN

   !Process the multibyte character and advance the pointer
   !by length of character:
   BEGIN

      !add code for processing a multibyte character
        .
        .

      !advance the pointer:
      @TESTMBCSCHAR := @TESTMBCSCHAR + $DBL(CHARSIZE.<8:15>);
   END;
   ELSE

   !Process single-byte character and advance pointer by one
   !byte:
   BEGIN

      !add code for processing single-byte character
        .
        .

      !Advance the pointer:
      @TESTMBCSCHAR := @TESTMBCSCHAR + 1D;
   END;
END;
```

Dealing With Fragments of Multibyte Characters

If a read operation of a text string of multibyte characters finishes when the specified read count is satisfied, then you cannot be sure whether the last byte read is the last byte of a character or the first byte of a multibyte character. If it is the first byte of a multibyte character, then its meaning is lost without the trailing byte. You should therefore call the MBCS_TRIMFRAGMENT_ procedure, which checks the validity of the last byte read and truncates it if it is the first byte of a multibyte character.

To use the MBCS_TRIMFRAGMENT_ procedure, you must supply it with a pointer to the text string and the length of the text string in bytes. For example:

```
INT BUFFER[0:79];                    !input buffer
STRING SBUFFER := @BUFFER '<<' 1;    !byte pointer to input
                                     ! buffer
 .
 .

CALL READ(BUFFER,RCOUNT,BYTES^READ);
IF <> THEN CALL DEBUG;

IF BYTES^READ = RCOUNT THEN
CALL MBCS_TRIMFRAGMENT_(@SBUFFER,
                        BYTES^READ);
```

On return, the bytes-read parameter specifies the number of bytes in the text string after the multibyte fragment is removed.

Handling Multibyte Blank Characters

Many applications expect an ASCII blank character (%H20) as a word delimiter in text strings. Multibyte character sets typically use a multibyte character to represent a blank. Some conversion therefore needs to be done if an application written for standard ASCII input is to work for multibyte character sets. This conversion is done using the MBCS_REPLACEBLANK_ procedure.

To use the MBCS_REPLACEBLANK_ procedure, you must supply it with a pointer to the text string to be converted and the length of the text string as follows:

```
CALL MBCS_REPLACEBLANK_(@SBUFFER,
                        BYTES^READ);
```

On return, the text buffer contains the same text as input except that any multibyte blank characters are converted to pairs of ASCII blanks. An application that expects ASCII blank characters can now process the text string correctly. At the same time, the integrity of the text string structure is maintained by using two ASCII blank characters to keep the text string the same length.

Determining the Character Size of a Multibyte Character Set

All currently supported multibyte character sets have two bytes per character. To prepare your programs for future expansion, however, you may need to know the character size. To find the character size of a multibyte character set, use the MBCS_CHARSIZE_ procedure.

To use the MBCS_CHARSIZE_ procedure, you must supply it with the number of the character set (as returned by the MBCS_CODESETS_SUPPORTED_ procedure). You receive the number of bytes per character in the return value:

```
RESULT := MBCS_CHARSIZE_(CHARACTER^SET);
```

Case Shifting With Multibyte Characters

Usually you can use the SHIFTSTRING procedure (or CASECHANGE or STRING_UPSHIFT_) to upshift or downshift a string of multibyte characters or multibyte characters mixed with single-byte (ASCII) characters. The following example upshifts a string provided in the TEXT^STRING buffer:

```
CASE^BIT := 0;                              !zero for upshifting
CALL SHIFTSTRING(TEXT^STRING,
                 BYTE^COUNT,
                 CASE^BIT);
```

As with all string-manipulation operations that involve multibyte characters, you must start your upshift or downshift operation on the first byte of a character. You can arrive at a first byte either by pointing to the first byte of a string or by using an MBCS_CHAR_ procedure call.

The SHIFTSTRING, CASECHANGE, or STRING_UPSHIFT_ procedures will work with multibyte characters, because your system is configured with versions of these procedures that work for your default character set. If you need to apply a string-shift operation to a string of text that does not belong to the default character set, you must instead use the MBCS_SHIFTSTRING_ procedure. The following call does the same thing as the SHIFTSTRING example above but for a different character set:

```
CASE^BIT := 0;                             !zero for upshifting
CHAR^SET := 9;                             !Tandem Hangul
CALL MBCS_SHIFTSTRING_(@TEXT^STRING, !string to upshift
                       BYTE^COUNT,   !number of bytes
                       CASE^BIT,
                       CHAR^SET);
```

Testing for Special Symbols

It is possible that special symbols used in a single-byte character set may appear as one byte of a multibyte character. It would be a mistake to interpret these bytes as single-byte special symbols. You should therefore test such a byte to see if it is part of a multibyte character.

To test for special symbols that are part of multibyte characters, you can use the MBCS_TESTBYTE_ procedure. First, you would scan the string for the special symbol, then call MBCS_TESTBYTE_ to check whether the byte is a single-byte character or part of a multibyte character.

To use the MBCS_TESTBYTE_ procedure, you must supply it with the buffer containing the string to be tested, the length of the buffer, and an index into the buffer identifying the byte to be tested. The following example scans a text string searching for a special character, then checks whether that byte is part of a multibyte character set:

```
SCAN SBUFFER UNTIL SPECIAL -> @SPECIAL^CHARACTER;

TEST^INDEX := @SPECIAL^CHARACTER - @SBUFFER;
RESULT := MBCS_TESTBYTE_(BUFFER,
                         BUFFER^LENGTH,
                         TEST^INDEX);
```

The value returned in RESULT indicates what happened:

If RESULT is...	Then the byte identified by the ***`test-index`*** parameter is...
0	A single-byte character
1	The first byte of a multibyte character
2	An intermediate byte (neither first or last byte) of a multibyte character
3	The last byte of a multibyte character

The *`testindex`* parameter contains the byte index to the first byte of the character.

Sample Program

The following program uses many of the system procedures that support multibyte characters. The program is similar to the program shown earlier in this section to illustrate the use of the FIXSTRING procedure. This example, however, implements an FC command for an environment that uses multibyte characters.

The enhancements made to the program shown below are as follows:

- The INITIALIZE^TERMINAL procedure uses the MBCS_CODESETS_SUPPORTED_ procedure to check whether multibyte code sets are supported. If not, then the program stops.
- The COMMAND^INTERPRETER procedure uses the MBCS_TRIMFRAGMENT_ procedure to check that the last bytes of the entered command are not a fragment of a multibyte character, and then uses the MBCS_REPLACEBLANK_ procedure to convert any multibyte blanks into double ASCII blank characters.
- The FC procedure also uses the MBCS_TRIMFRAGMENT_ and MBCS_REPLACEBLANK_ procedures to process the string again after editing.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!Global literals and variables:

INT     TERM^NUM;                        !open terminal file number

STRUCT .CI^STARTUP;                      !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULT;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOLUME[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAM[0:529];
END;

LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;
LITERAL ABEND   = 1;

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(WRITEX,WRITEREADX,
?  FILE_OPEN_,FIXSTRING,PROCESS_STOP_,INITIALIZER,
?  SHIFTSTRING,MBCS_CODESETS_SUPPORTED_,MBCS_TRIMFRAGMENT_,
?  MBCS_REPLACEBLANK_,OLDFILENAME_TO_FILENAME_)
?LIST
```

```
!--------------------------------------------------------------
! Integer procedure edits the command buffer and returns 1
! if edited command should be executed.  This procedure
! allows the user a change of mind about editing the command
! by returning 0.
!--------------------------------------------------------------

INT PROC FC(COMMAND,LAST^COMMAND,NUM,SAVE^NUM);
STRING .COMMAND;
STRING .LAST^COMMAND;
INT    .NUM;
INT    .SAVE^NUM;

BEGIN
   STRING  .TEMPLATE^ARRAY[0:71]; !template used for edit
                                  ! changes
   INT      TEMPLATE^LENGTH;      !length of template
   INT      MAX^LEN := 8;         !maximum length of edited
                                  ! command
   STRING  .BUFFER[0:71];         !I/O buffer
   STRING  .S^PTR;                !pointer to end of text
                                  ! string

!  Set command prompt to "< ":

   COMMAND[-2] ':=' "< ";

!  Set NUM equal to size of previous command:

   NUM := SAVE^NUM;

!  Put previous command in command buffer:

   COMMAND ':=' LAST^COMMAND FOR NUM;

!  Edit the command each time through the loop. The loop
!  enables the user to check the results of an edit and then
!  edit again if necessary:

   DO
   BEGIN

   !  Write the command to be edited to the terminal:

      CALL WRITEX(TERM^NUM,COMMAND[-2],NUM + 2);
      IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                    !specifier!,
                                    ABEND);
```

```
!  Set the FC prompt to " ." and read template typed by
!  user:

   TEMPLATE^ARRAY ':=' " .";
   CALL WRITEREADX(TERM^NUM,TEMPLATE^ARRAY,2,72,
                   TEMPLATE^LENGTH);

!  If WRITEREADX returns an error, or if the template
!  contains exactly two slashes, then return with no
!  changes:

   IF > OR TEMPLATE^LENGTH = 2
      AND TEMPLATE^ARRAY = "//" THEN
   BEGIN
      NUM := SAVE^NUM;
      COMMAND ':=' LAST^COMMAND FOR NUM;
      RETURN 0;
   END;

!  Otherwise call FIXTRING to alter the command according
!  to the instructions in the template:

   CALL FIXSTRING(TEMPLATE^ARRAY,TEMPLATE^LENGTH,COMMAND,
                  NUM,MAX^LEN);
   IF > THEN
   BEGIN

   !  The replacement string is greater than MAX^LEN, so
   !  print a diagnostic message and return to beginning
   !  of loop:

      BUFFER ':=' "Replacement string too long "
                  -> @S^PTR;
      CALL WRITEX(TERM^NUM,BUFFER,@S^PTR '-' @BUFFER);
      IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                    !specifier!,
                                    ABEND);
   END
   ELSE IF < THEN CALL PROCESS_STOP_(!process^handle!,
                                     !specifier!,
                                     ABEND);

!  Trim multibyte fragments from end of string:

   CALL MBCS_TRIMFRAGMENT_(@COMMAND,NUM);

!  Replace multibyte blanks with two ASCII blank
!  characters:

   CALL MBCS_REPLACEBLANK_(@COMMAND,NUM);
```

```
      !  Upshift all characters in the edited command in case
      !  any characters were typed in lowercase (assumes default
      !  multibyte character set; otherwise you would need to
      !  use MBCS_CHAR_ and MBCS_SHIFTSTRING_):

         CALL SHIFTSTRING(COMMAND,NUM,0);
      END

   !  Loop until user responds to FC prompt with a carriage
   !  return only:

      UNTIL NOT TEMPLATE^LENGTH;

   !  Return to command interpreter to execute edited command:

      RETURN 1;

   END;
```

```
!-------------------------------------------------------------
! Procedure prompts the user for a command and then processes
! the command.  This procedure loops indefinitely until the
! user types the EXIT command.
!-------------------------------------------------------------

PROC COMMAND^INTERPRETER;
BEGIN
   STRING .LAST^COMMAND[0:7];     !buffer for last command
   INT     NUM;                   !number of bytes
                                  ! transferred
   INT     SAVE^NUM;              !previous number of bytes
                                  ! transferred
   STRING .COMMAND[-2:7] := "< "; !command buffer
   STRING .BUFFER[0:71];          !I/O buffer
   STRING .S^PTR;                 !string pointer

   INT REPEAT := 0;               !when 0, prompt for new
                                  ! command; when 1,
                                  ! execute repaired command

!  Loop until user types "EXIT":

   WHILE 1 DO
   BEGIN

   !  If repeat not set, prompt user for a new command:

      IF NOT REPEAT THEN
      BEGIN
         COMMAND[0] := " ";
         COMMAND[1] ':=' COMMAND[0] FOR 7;
         COMMAND ':=' "< ";
         CALL WRITEREADX(TERM^NUM,COMMAND,2,9,NUM);
         IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                       !specifier!,
                                       ABEND);
      END;

   !  Trim multibyte fragments from end of string:

      CALL MBCS_TRIMFRAGMENT_(@COMMAND,NUM);

   !  Replace multibyte blanks with two ASCII blank
   !  characters:

      CALL MBCS_REPLACEBLANK_(@COMMAND,NUM);
      !  Upshift all characters in the edited command in case
      !  any characters were typed in lowercase (again assuming
      !  default multibyte character set:

         CALL SHIFTSTRING(COMMAND,NUM,0);

      !  If the command is "FC" then call the FC procedure,
      !  returning 1 if the fix is accepted or 0 if it is not.
      !  If the command is EXIT, then stop the program.
      !  If the command is any other valid command, then process
      !  the command (this program simply displays the command
      !  name).  If an illegal command, then print a diagnostic
      !  message:

         IF COMMAND = "FC"
            THEN REPEAT := FC(COMMAND,LAST^COMMAND,NUM,SAVE^NUM)

         ELSE BEGIN
           IF COMMAND = "EXIT" THEN CALL PROCESS_STOP_

            ELSE IF COMMAND = "COMMAND1"
               THEN CALL WRITEX(TERM^NUM,COMMAND,NUM)

            ELSE IF COMMAND = "COMMAND2"
               THEN CALL WRITEX(TERM^NUM,COMMAND,NUM)

            ELSE BEGIN
               BUFFER ':=' COMMAND FOR 8;
               BUFFER[8] ':=' ":  Illegal Command " -> @S^PTR;
               CALL WRITEX(TERM^NUM,BUFFER,@S^PTR '-' @BUFFER);
               IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                              !specifier!,
                                              ABEND);
            END;

         !  Reset the repeat flag:

            REPEAT := 0;
         END;

      !  If the command length is nonzero, then save it in the
      !  LAST^COMMAND array for possible editing by a subsequent
      !  FC command:

         IF NUM THEN
         BEGIN
            SAVE^NUM := NUM;
            LAST^COMMAND ':=' COMMAND FOR SAVE^NUM;
         END;
      END;
   END;
```

```
!-------------------------------------------------------------
! Procedure to save the Startup message in a global
! structure.
!-------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,
             MESSAGE,LENGTH,MATCH) VARIABLE;
INT .RUCB,
    .START^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;

END;
```

```
!------------------------------------------------------------
! Procedure to open the terminal file specified in the IN
! file of the Startup message and check that multibyte
! character sets are supported.  The program stops if
! multibyte character sets are not supported.
!------------------------------------------------------------

PROC INITIALIZE^TERMINAL;
BEGIN

   INT(32)     RESULT;
   STRING     .S^PTR;
   STRING     .BUFFER[0:71];
   STRING     .TERM^NAME;
   INT         TERMLEN;
   INT         ERROR;

!  Read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
      CI^STARTUP.INFILE.VOLUME,
      TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);

!  Check that multibyte characters are supported:

   RESULT := MBCS_CODESETS_SUPPORTED_;
   IF RESULT = 0D THEN
   BEGIN
      BUFFER ':=' "Character Set Not Supported" -> @S^PTR;
      CALL WRITEX(TERM^NUM,BUFFER,@S^PTR '-' @BUFFER);
      IF <> THEN CALL PROCESS_STOP_(!process^handle!,
                                    !specifier!,
                                    ABEND);
   END;

END;
```

```
!------------------------------------------------------------
! Main procedure initializes the IN file and then calls the
! command interpreter.
!------------------------------------------------------------

PROC INITIALIZE MAIN;
BEGIN

!  Initialize the IN file:

   CALL INITIALIZE^TERMINAL;

!  Call the command interpreter:

   CALL COMMAND^INTERPRETER;

END;
```

20

Interfacing With the ERROR Program

The ERROR process returns error-message text associated with a file-system error number. You can access the ERROR process in one of the following ways:

- By typing the ERROR command and a file-system error number in response to the command-interpreter prompt. The ERROR process responds by displaying the error number and associated text on the terminal or designated OUT file.
- Programmatically by sending the error-message number to the ERROR process in an interprocess message.

For details on using the ERROR program with the TACL program, refer to the *Guardian User's Guide*. The remainder of this section discusses how to programmatically interact with the ERROR process from your own application.

To obtain the message text for a file-system error number, your program must execute the following sequence:

1. Start an ERROR process using the PROCESS_CREATE_ procedure.
2. Open the ERROR process.
3. Send a Startup message to the ERROR process using the WRITE procedure. In this Startup message, you must specify $RECEIVE as the OUT file and the ASCII code for the error number in the parameter string.
4. Read and process the error message. Use the WRITEREAD procedure for reading message text.
5. Close and delete the ERROR process.

The following subsections describe each of the above operations in detail. The sample program at the end of this section shows one way of writing an application to access error-message text. Advanced readers may prefer to go straight to the sample program.

Creating an ERROR Process

To create the ERROR process, you can use the PROCESS_CREATE_ procedure. You can create the process in a waited or nowait manner. The following example creates a named ERROR process and waits for the creation to finish:

```
LITERAL MAXPDLEN = ZSYS^VAL^LEN^PROCESSDESCR;
STRING  OBJFILENAME;                 !object file name
INT    .PROCESS^HANDLE[0:9],         !process handle of ERROR
                                     ! process
        OBJFILENAME^LENGTH,          !length of ERROR file name
        NAME^OPTION;                 !specifies want process
                                     ! named by system
STRING .PROCESS^DESCR[0:MAXPDLEN -1],!process descriptor
INT     PROCESS^DESCR^LEN,           !length of process
                                     ! descriptor
        ERROR,                       !error return from
                                     ! PROCESS_CREATE_
  .
  .

OBJFILENAME ':=' "$SYSTEM.SYSTEM.ERROR" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR '-' @OBJFILENAME;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEDBYSYS;
ERROR := PROCESS_CREATE_(OBJFILENAME:OBJFILENAME^LENGTH,
                         !library^filename:library^file^len!,
                         !swap^filename:swap^file^len!,
                         !ext^swap^file^name:ext^swap^len!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         !name:length!,
                         PROCESS^DESCR:MAXPDLEN,
                         PROCESS^DESCR^LEN);
IF ERROR^RETURN <> 0 THEN ...
```

The following example creates an ERROR process, initiating the creation in a nowait manner. The Process create message (message -102) is delivered to the $RECEIVE file when the creation is complete. This message contains the process handle and process descriptor of the created process.

```
INT    .BUFFER[0:BUFSIZE];
STRING .SBUFFER ':=' @BUFFER '<<' 1;

OBJFILENAME ':=' "$SYSTEM.SYSTEM.ERRORX" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR '-' @OBJFILENAME;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEDBYSYS;
NOWAIT^TAG := 1D;
```

```
ERROR := PROCESS_CREATE_(OBJFILENAME:OBJFILENAME^LENGTH,
                         !library^filename:library^file^len!,
                         !swap^filename:swap^file^len!,
                         !ext^swap^file^name:ext^swap^len!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         !name:length!,
                         PROCESS^DESCR:MAXPDLEN,
                         PROCESS^DESCR^LEN,
                         NOWAIT^TAG);
IF ERROR <> 0 THEN ...
 .
 .
CALL READUPDATEX(RCV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF <> THEN ...

IF BUFFER[0] = -102 THEN                !process create
BEGIN                                   ! completion message
   IF BUFFER [13] <> 0 THEN ...         !error
   ELSE
   BEGIN
      NOWAIT^TAG := BUFFER[1] FOR 2;
      PROCESS^HANDLE ':=' BUFFER[3] FOR 10;
      PROCESS^DESCRIPTOR^LENGTH := BUFFER[15];
      PROCESS^DESCRIPTOR ':=' BUFFER[20] FOR
                              PROCESS^DESCRIPTOR^LENGTH;
   END;
END;
```

For more details on creating processes and on the PROCESS_CREATE_ completion message, refer to Section 16, Creating and Managing Processes.

Opening an ERROR Process

You open an ERROR process as you would any process by passing the process name or process descriptor to the FILE_OPEN_ procedure:

```
CALL FILE_OPEN_(PROCESS^NAME:PROCESS^NAME^LENGTH,
                PROCESS^FILE^NUMBER);
```

See Section 2, Using the File System, for details on opening process files.

Sending an ERROR Process a Startup Message

After opening the ERROR process file, you must use the WRITE[X] procedure to send the ERROR process a Startup message. This Startup message must contain the following information:

- -1 in the first word to identify the message as the Startup message.
- The $RECEIVE file as the OUT file. Doing so causes the ERROR process to send its output to your process (the process that opened it).
- The error number in ASCII code in the parameter string. The ERROR process expects to find the error number to process in the parameter string.

The Startup message sent to the ERROR process will therefore be similar to the following:

	Word								
MSG^CODE	0	-1							
DEFAULT.VOLUME	1								
DEFAULT.SUBVOL	5								
INFILE.VOLUME	9								
INFILE.SUBVOL	13								
INFILE.FNAME	17								
OUTFILE.VOLUME	21	"$"	"R"	"E"	"C"	"E"	"I"	"V"	"E"
OUTFILE.SUBVOL	25								
OUTFILE.FNAME	29								
PARAMS	33	6	0	null					

VST102.VSD

Note that the default volume and subvolume and IN file information are not required by the ERROR program.

Once you have formed the Startup message, send it to the ERROR process using the WRITE[X] procedure and the file number returned when you opened the ERROR process:

```
BUFFER := -1;
BUFFER[21] ':=' ["$RECEIVE", 8 * ["  "]];!OUT file
BUFFER[33] := ERROR^NUMBER;              !parameter string
BUFFER[34] := 0;

CALL WRITEX(PROCESS^FILENUMBER,SBUFFER,70);
```

Reading and Processing Error-Message Text

To read the error-message text, you issue a WRITEREAD[X] procedure call against the ERROR process. Because most messages need more than one line of information, you need to issue the WRITEREAD[X] procedure several times to read the entire message text. The ERROR process returns an end-of-file indication when you reach the end of the message, causing WRITEREAD[X] to return a greater than (>) condition code.

The arrival of a user message on $RECEIVE is enough to tell the ERROR process to reply with the next line of the message. Therefore, you need to send only a zero length string with each WRITEREAD[X] call.

Once you have read the message text, you can process it in any way you like. Typically, you will print the message text on the terminal or possibly save it in a log file.

The following code fragment reads a message from the ERROR process and prints it on the home terminal:

```
!Loop while ERROR process still sending text:
EOF := 0;
WHILE NOT EOF DO
BEGIN

   !Read the error message text into the buffer:
   CALL WRITEREADX(PROCESS^FILENUMBER,SBUFFER,0,132,
                  BYTES^READ);

   !Set flag if end of message text:
   IF > THEN EOF := 1;

   !Print buffer if not end of message text:
   ELSE
   BEGIN
      CALL WRITEX(TERM^NUM,SBUFFER,BYTES^READ);
      IF <> THEN ...;
   END;
END;
```

Refer to Section 16, Creating and Managing Processes, for more details on sending and receiving the Startup message, or to Section 8, Communicating With a TACL Process, for a detailed discussion of the structure of the Startup message.

Closing and Deleting an ERROR Process

Once you have read the error-message text, you can close an ERROR process file with the FILE_CLOSE_ procedure and stop an ERROR process with the PROCESS_STOP_ procedure. For example:

```
CALL FILE_CLOSE_(PROCESS^FILENUMBER);

CALL PROCESS_STOP_(PROCESS^HANDLE);
```

Using the ERROR Process: An Example

The following sample program interfaces with the ERROR process to print file-system error messages on the home terminal. The example is made up of the following procedures:

- The PRINT^ERROR procedure is called by the FILE^USER (main) procedure whenever a file-system error occurs. This procedure takes one formal parameter: the file number of the file against which the error occurred. PRINT^ERROR obtains the corresponding error number using the FILE_GETINFO_ procedure and then passes that number to the ERROR process. Finally, it displays the returned text on the home terminal.
- FILE^USER is the main procedure. It calls the INIT procedure to initialize the terminal and then calls the PRINT^ERROR procedure, passing it a file number.
- The INIT and SAVE^STARTUP^MESSAGE procedures open the IN file as specified in the Startup message.

```
?INSPECT,SYMBOLS,NOMAP,NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

LITERAL BUFSIZE = 128;
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;
LITERAL MAXPDLEN = ZSYS^VAL^LEN^PROCESSDESCR;
LITERAL ABEND   = 1;

!Global variables:
INT     TERMNUM;                 !terminal file number
INT     FNUM;                    !generic file number
STRING .S^PTR;
STRING SBUFFER[0:BUFSIZE];

STRUCT .CI^STARTUP;              !Startup message, used not only
BEGIN                            ! to receive Startup message
   INT MSGCODE;                  ! from creator but also to
   STRUCT DEFAULT;               ! send Startup message to
   BEGIN                         ! ERROR process and receive
      INT VOLUME[0:3];           ! reply
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILEID[0:3];
   END;
   STRING PARAMS[0:529];
END;
STRING .S^STARTUP := @CI^STARTUP[0] '<<' 1;

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,
?       PROCESS_CREATE_,WRITEX,WRITEREADX,PROCESS_STOP_,
?       FILE_CLOSE_,OLDFILENAME_TO_FILENAME_,FILE_GETINFO_)
?LIST
```

```
!------------------------------------------------------------
! Procedure to print error text on the home terminal.  Runs
! and opens the error program, sends it an error number and
! receives the error-message text.
!------------------------------------------------------------

PROC PRINT^ERROR(FNUM);
INT FNUM;                          !file number of file with
                                   ! error against it

BEGIN
   INT    .PROCESS^HANDLE[0:9];     !process handle of ERROR
                                   ! process
   STRING .OBJ^FNAME[0:MAXFLEN - 1];!object file name
   INT     NAME^OPTION := 2;        !specifies a system-named
                                   ! process
   STRING .PROC^DESCR[0:MAXPDLEN - 1];!process descriptor
   INT     PROC^DESCR^LEN;          !length of process
                                   ! descriptor
   INT     PROCNUM;                 !process file number
   INT     ERROR^RETURN;            !error return from
                                   ! PROCESS_CREATE_
   INT     EOF;                     !indicates end of message
                                   ! text
   INT     COUNT^READ;
   INT     ERROR^NUMBER;            !number of file-system
                                   ! error to display!
   INT     ERROR;                   !local file-system error

!  Get the file-system error to display:

   CALL FILE_GETINFO_(FNUM,ERROR^NUMBER);

!  Create the ERROR process:

   OBJ^FNAME ':=' "$SYSTEM.SYSTEM.ERRORX" -> @S^PTR;
   ERROR^RETURN := PROCESS_CREATE_(
                        OBJ^FNAME:@S^PTR '-' @OBJ^FNAME,
                        !library^filename:library^file^len!,
                        !swap^filename:swap^file^len!,
                        !ext^swap^file^name:ext^swap^len!,
                        !priority!,
                        !processor!,
                        PROCESS^HANDLE,
                        !error^detail!,
                        NAME^OPTION,
                        !name:length!,
                        PROC^DESCR:MAXPDLEN,
                        PROC^DESCR^LEN);
   IF ERROR^RETURN <> 0 THEN
   BEGIN
      SBUFFER ':=' "Unable to create Error process. "
                   -> @S^PTR;
      CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
      CALL PROCESS_STOP_(!process^handle!,
                         !specifier!,
                         ABEND);
   END;

!  Open the ERROR process:

   ERROR := FILE_OPEN_(PROC^DESCR:PROC^DESCR^LEN,PROCNUM);
   IF ERROR <> 0 THEN
   BEGIN
      SBUFFER ':=' "Unable to open Error process. "
                   -> @S^PTR;
      CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);
      CALL PROCESS_STOP_(PROCESS^HANDLE,
                         !specifier!,
                         ABEND);
      CALL PROCESS_STOP_(!process^handle!,
                         !specifier!,
                         ABEND);
   END;

!  Blank the Startup message:

   CI^STARTUP.MSGCODE[0] ':=' "  ";
   CI^STARTUP.MSGCODE[1] ':=' CI^STARTUP.MSGCODE[0] FOR 34;

!  Format the Startup message:

   CI^STARTUP.MSGCODE := -1;
   CI^STARTUP.OUTFILE.VOLUME ':='
      ["$RECEIVE", 8 * ["  "]];           !OUT file
                                          ! parameter string
   CI^STARTUP.MSGCODE[33] := ERROR^NUMBER;
   CI^STARTUP.PARAMS[2] := 0;
   CI^STARTUP.PARAMS[3] := 0;

!  Send Startup message to ERROR program:

   CALL WRITEX(PROCNUM,S^STARTUP,70);
```

```
!  Loop while ERROR program still sending text:

   EOF := 0;
   WHILE NOT EOF DO
   BEGIN

   !  Read the error-message text into the buffer:

      CALL WRITEREADX(PROCNUM,S^STARTUP,0,132,COUNT^READ);

      !Set flag if end of message text:
      IF > THEN EOF := 1;

      !Print buffer if not end of message text:

      ELSE
      BEGIN
         CALL WRITEX(TERMNUM,S^STARTUP,COUNT^READ);
         IF <> THEN
         BEGIN
            SBUFFER ':='
               "Unable to communicate with Error process.  "
               -> @S^PTR;
            CALL WRITEX(TERMNUM,SBUFFER,
                        @S^PTR '-' @SBUFFER);
            CALL PROCESS_STOP_(PROCESS^HANDLE,
                               !specifier!,
                               ABEND);
            CALL PROCESS_STOP_(!process^handle!,
                               !specifier!,
                               ABEND);
         END;
      END;
   END;

!  Close the ERROR process file:

   CALL FILE_CLOSE_(PROCNUM);

!  Stop the ERROR process:

   CALL PROCESS_STOP_(PROCESS^HANDLE);
END;
```

```
!------------------------------------------------------------
! Procedure to copy the Startup message into a global
! structure.
!------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH,MATCH)VARIABLE;
INT .RUCB;
INT .START^DATA;
INT .MESSAGE;
INT  LENGTH;
INT  MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;


!------------------------------------------------------------
! Procedure to perform initialization for the program. It
! calls INITIALIZER to read and copy the Startup message into
! the global data area and then opens the IN file specified
! in the Startup message.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING  .TERM^NAME[0:MAXFLEN - 1];
   INT      TERMLEN;
   INT      ERROR;

!  Read and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
                         CI^STARTUP.INFILE.VOLUME,
                         TERM^NAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
END;
```

```
!-------------------------------------------------------------
! Main procedure initializes the home terminal and calls the
! PRINT^ERROR procedure.
!-------------------------------------------------------------

PROC FILE^USER MAIN;
BEGIN

!  Read Startup message:

   CALL INIT;
     .
     .

!  If file-system error:

   CALL PRINT^ERROR(FNUM);

END;
```

21 Writing a Requester Program

Recall from Section 1, Introduction to Guardian Programming, that a requester/server design has several advantages over the monolithic or unified program approach. Specifically, requesters and servers provide modularity by allowing the requester program to handle terminal I/O, while a server process provides database service. Such a model makes it easy to provide additional service by adding a new server process or to support a larger user community by duplicating requesters.

This section describes the various functions that are usually performed by a requester program. A programming example at the end of this section illustrates these functions using many of the procedures and techniques described in Sections 2 through 20 of this manual. This section thereby provides a summary.

This section should be read with Section 22, Writing a Server Program, which provides information about writing a server program. Section 22 provides three sample server programs that interact with the requester described in this section.

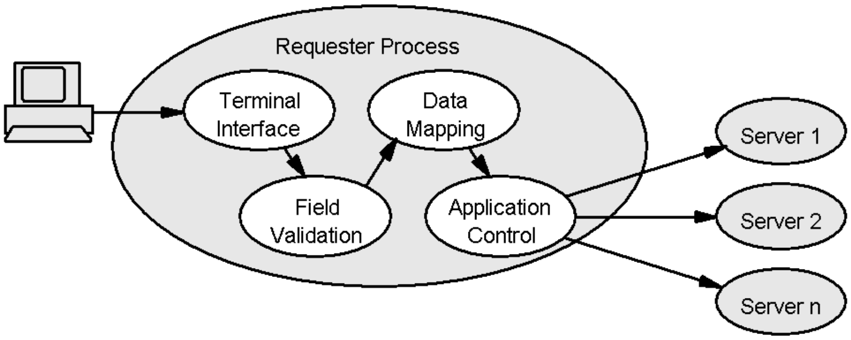
Functions of a Requester

In a typical requester/server application, the requester handles all terminal I/O while leaving the “back-end” database handling to a server process. The job of such a requester process can be broken down into the following functions (see Figure 21-1):

- Terminal interface
- Field validation
- Data mapping
- Application control

The following paragraphs describe each of these functions.

Figure 21-1. Functions of a Requester Process



VST103.VSD

Terminal Interface

The terminal interface deals with writing information to the terminal and reading information typed at the terminal. In its simplest form, the terminal interface consists of a series of prompts and user responses (using the WRITEREAD[X] procedure) to find out what the user wants the application to do. After the application has processed a request, the terminal-interface code usually displays the result on the terminal (using the WRITE[X] procedure).

Interaction with the terminal can be in conversational mode or block mode. See Section 10, Communicating With Terminals, for details.

Figure 21-1 shows a requester interface with just one terminal. To work with more than one terminal, either you can create duplicate requester processes, your requester can be programmed to interface with more than one terminal, or your requester can communicate with a terminal-simulation process that controls several terminals. Section 24, Writing a Terminal Simulator, discusses terminal-simulation processes.

Field Validation

The field-validation part of the requester checks the credibility of data entered at the terminal. For example, it might check that a person's name consists of alphabetic characters or that a person’s age is not more than 150 years.

Data Mapping

Data mapping involves conversion and structuring of input data into a form suitable for processing by the server process. For example, numeric ASCII input gets converted into signed binary numbers (using the NUMIN procedure) or the formatter converts input into a specific format for processing. See Section 19, Formatting and Manipulating Character Data, for details.

For the convenience of the server, the input data is usually placed in a data structure that enables the server to receive the data in known format.

Similarly, the requester usually needs to perform data mapping on information sent to the requester by the server before printing it on the terminal. Here, the requester usually receives the information in the form of a data structure. The requester must extract the information it needs from the data structure and, if necessary, convert it into a humanly readable form before writing it to the terminal.

Application Control

Application control is the part of the requester that interacts with the server process. This part of the requester can provide the following functions:

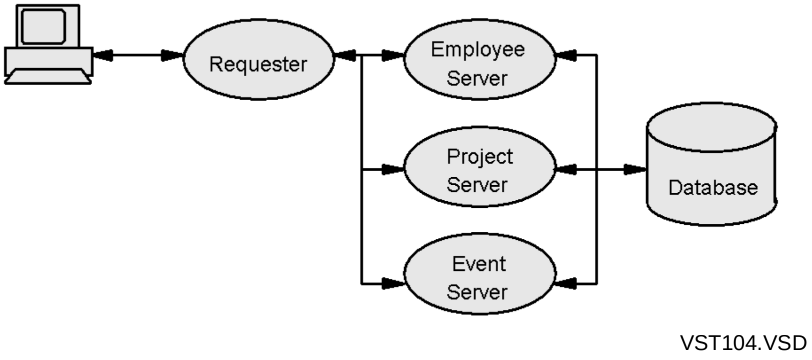
- Selection of a specific server process to do a specific job
- Selection of a generic context-free server
- Transaction control through the use of procedure calls to the TMF subsystem

The following paragraphs describe these functions.

Selecting a Server by Function

Typically, the application-control part of the requester process selects a server process to carry out a task dependent on the input provided by the user. For example, the requester could select one server process to print out a bank statement or a different server process to transfer money between accounts. Figure 21-2 shows another example.

Figure 21-2. Server Selection by Function

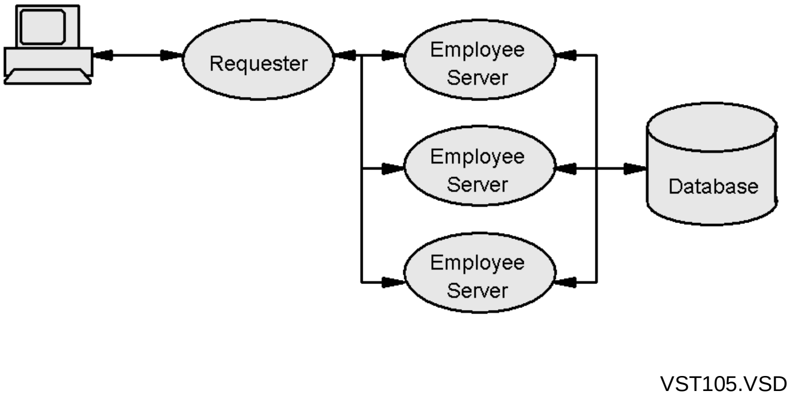


The requester establishes communication with the server of its choice by sending it the message that was formulated during the data-mapping phase. Usually, the requester expects a reply from the server and therefore sends the message to the server using the WRITEREAD[X] procedure. If no reply data is expected, you can use the WRITE[X] procedure. Section 6, Communicating With Processes, provides details on how to do this.

Selecting a Generic Server

Your requester process may choose from several functionally identical servers. For example, if a server process is heavily used, your application may share the load by running several functionally identical servers. In this kind of design, however, the servers must be context free so that the requester can select any available server. Figure 21-3 shows the model.

Figure 21-3. Selecting From Functionally Identical Servers



See Section 22, Writing a Server Program, for a discussion of context-free servers.

Transaction Control Using TM/MP

The NonStop Transaction Manager/MP (TM/MP) provides the ability to control transactions against a database from the requester process. These techniques ensure data integrity by controlling access to the database from the requester process. See the *Introduction to NonStop Transaction Manager/MP (TM/MP)* for an overview or the *NonStop TM/MP Application Programmer’s Guide* for programming information.

File System I/O Synchronization

The Guardian file system provides a synchronization mechanism for detecting lost or duplicate interprocess messages. This mechanism becomes important when dealing with nonretryable I/Os that are not audited (protected by TMF transactions). Synchronization is done by the participating processes; however, the file system might automate the path-failure handling for the requester.

Note that processes that open $RECEIVE and other files are considered both requesters and servers in the context of this discussion.

- A process is a server while processing a message received through $RECEIVE.
- A process is a requester while performing I/O on other files it has opened.

The FILE_OPEN_ parameter *sync-depth* is used to tell the file system and the server important information about how an opened file must be accessed.

Sync-Depth

The *sync-depth* parameter has several functions, depending on the device type, but its most important function is to tell the server how many responses it needs to save.

Each opened process file has a 32-bit sync ID value that the file system automatically increments by one for each nonretryable operation issued (for example, in WRITE or WRITEREAD operations). Each I/O request has a sync ID value that enables the server to detect duplicate requests. A server can retrieve the sync ID by calling FILE_GETRECEIVEINFO_. For each opener, the server saves the result and the sync ID of the last request that was successfully processed. When a new request is read from $RECEIVE, the server compares the sync ID value with its last processed sync ID. If the new sync ID is less than or equal to the value of the saved sync ID value, the request is a duplicate.

Servers must be written to handle sync depth and sync ID values. For example, a request is too old if the new sync ID minus the saved sync ID is greater than or equal to the sync depth value. This situation can occur as a result of an application error, which the server must be coded to handle. Some high-level languages, like COBOL85, handle sync depth processing automatically. A new request always has a sync ID value that is equal to the saved sync ID value plus 1.

Sync-depth value	**Description**
0	Any error is immediately returned with no retry attempted. WRITE operations of disk files are not checkpointed.
> 0 (greater than 0)	Enables the file system to automatically retry any path error that might occur (file-system errors 200 through 211). Path error retries are transparent to the requester. For disk files, checkpoints of WRITE operations are enabled.
n	The number of nonretryable operations the requester might perform between file sync block checkpoints. To handle this, the server must “remember” or save the *n* most recent responses. The server must handle a duplicate request that is up to *n* requests old by immediately returning a saved response corresponding to the original request.

Sync-Depth in Practice

The following is the sequence of events that occurs for a path error retry, which is performed automatically if the sync depth value is greater than 0.

- The requester sends an I/O to the server. The file system increments the sync ID for the file and locates the server using the process handle determined at open time. The request is then sent to the server process.
- Consider a scenario where the request fails with a path error because the server switched to its backup process. The switch causes the process pair's PPD entry to be updated, because the primary server process stopped and the backup server process assumed the primary role. In this case, the following occurs:
    - The request is sent to the server.
    - A path error occurs because the server's primary process fails. This causes the server to switch to its backup process.
    - The file system checks the sync depth and determines that the request must be retried automatically (sync depth value is greater than 0). If sync depth value is 0, the path error is returned to the requester.
    - Because the sync depth value is greater than 0, the file system resends the request, this time to the backup process, which has taken over. The sync ID is not incremented.
    - The server uses sync IDs to determine how to handle the request:
        - If the sync depth value is less than the sync ID minus the last saved sync ID, then the request is too old. The server returns an error indicating that the request is too old.
        - If the sync ID is otherwise less than or equal to the last saved sync ID, then it is a duplicate request. The corresponding saved result is returned immediately. If the server is multithreaded, the original request might not be completed, in which case the old request tag is released and replaced with the new request tag.
        - If the sync ID is greater than the last saved sync ID, then it is a new request. The request is processed normally. When the response occurs, it is saved together with the sync ID.
    - The first time the server received the request, it was treated as a new request. The second time the same request is received, it might be a duplicate request or a new request, depending on at which point the server's backup process resumed processing. If it is a duplicate request, the server replies with the saved response, indicating a successful I/O completion.
    - The requester I/O finishes.

Because the sync depth value is greater than 0, there is no indication in the requester that a path retry has been performed.

File Sync Block Checkpoints (Example)

Assume that a requester is a process pair. The process pair opens a disk file or a process with sync depth value of 3. This causes the server to allocate three status blocks for the saved results with the opener control block.

In its main loop, the requester performs a single CHECKPOINT call, including the file sync block. The call to CHECKPOINT is followed by three WRITE operations on the file. This is repeated for each iteration of the loop in the requester. The server saves the results of the WRITE operations in its three consecutive status blocks. Note that if the requester issues a fourth WRITE operation without an intervening file sync block checkpoint, the server no longer retains information about the first WRITE operation (the requester is violating its sync depth). The server cannot detect this error during normal processing. The error is detected only if the requester fails while performing its fourth WRITE operation, because the sync ID of the first WRITE operation is too old for a retry.

If the requester’s primary process fails, the backup process takes over at the most current checkpoint, which is just before the three WRITE operations, no matter where the primary process actually was executing at the time of the failure. The backup process becomes the primary process. As the process reiterates up to and past the actual point of failure it might redo all or none of these WRITEs. The server might receive duplicate requests. There are four possible outcomes:

- No WRITE operations were processed by the primary process. In this case, no duplicates are detected by the server. Processing continues normally.
- WRITE 1 was processed by the primary process. The server recognizes the duplicate request. (Its last sync ID is that of the first WRITE executed by the primary process before it failed.) The result is returned without reexcuting the WRITE operation in the server. When the new primary process executes WRITE 2, it becomes a new request and it is processed normally by the server. The same is true for WRITE 3.
- WRITEs 1 and 2 were processed by the primary process. These WRITE operations are recognized and handled as duplicates. WRITE 3 becomes a new request.
- All three WRITE operations were successfully processed by the primary process before the failure occurred. The new primary process gets the saved responses from the server. On the next iteration of the main loop, the sync ID values tell the server that these are new requests.

Thus, the server process ensures that any failure in the requester does not result in WRITE operations being executed more than once by the server.

I/O Synchronization in Requester

Normally, there are minimal synchronization requirements on the requester side, as long as the requester is not a process pair. As a rule of thumb, set the values for *nowait-depth* and *sync-depth* as follows:

- *Nowait-depth* to either 0 or 1, depending on your choice of waited or no-waited I/O.
- *Sync-depth* to 1. *Sync-depth* can be set to 0, but this requires the requester to handle path retries.

If the requester is a process pair, it is important to ensure that any I/O that is resent on a takeover is using the same sync ID as the original request. Requester process pairs might also use sync depth values greater than 1 to optimize checkpointing.

I/O Synchronization in Server

On the server side, there are no automatic recovery mechanisms for path failures. Servers are responsible for keeping track of their requests, except in COBOL85 programs. COBOL85 has Guardian-specified extensions that allow it to effectively handle openers and I/O synchronization.

This problem can be avoided by writing context free servers and using the TMF transactions for retries.

Path failures normally cause requesters to resend pending requests. In servers, these resends must be detected as duplicate requests. The following are typical path failure scenarios that servers must handle:

- If the server is a process pair, it must handle duplicate requests whenever it switches processing to its backup process.
- If a requester (opener) is a process pair, the server might receive a duplicate request at any time, because the requester backup process took over.
- If any of the requesters are in other systems, messages might be resent because of communication failures between systems.

The information needed to track openers and requests is found in the OPEN system messages and in data returned by FILE_GETRECEIVEINFO_ calls. The data is normally collected into an open control block for each opener.

The server needs to manage each opener separately and save responses for up to the greater of the open’s nowait depth and sync depth values, in order for it to be fault tolerant. Note that a process pair normally maintains two opens for any given file, one from the primary process and one from the backup process. The primary process first opens the file and then instructs the backup process to do the same through a call to FILE_OPEN_CHKPT_. This is known as a paired or a logical open.

The server must call MONITORCPUS (and MONITORNET if requesters reside in other systems) to detect failing CPUs. If a requester resides in a failing CPU, no close message is received. Instead, when a “processor down” system message is received, the server must check all open control blocks for requesters in that CPU and implicitly close those opens.

In order to properly manage an opener, the server needs an open control block containing the following information:

- The requesters’ process name

- The requesters' primary process handle
- The requesters' backup process handle
- The file numbers the requester used for the opens (each open has a distinct file number, although normally the backup's file number is the same as the primary's file number)
- The *sync-depth* value of the open
- The last sync ID value received
- Buffers to store responses (sync ID, error code, reply size, and reply data)

The primary and backup process handles are subject to change during the lifetime of the open. The process handles can change if one or more of the following occurs:

- CPU or process failures. If the primary CPU fails, the backup's process handle must be copied into the primary's process handle slot and the backup's process handle must be reset. If the backup CPU fails, its process handle must be reset. Notification of CPU failures are obtained using the MONITORCPUS and the MONITORNET procedures. Process failures cause a close to be received from either the primary process or the backup process. Note that a request can be received from the backup process before you receive a close from the primary process or a CPU failure message, depending on the timing.
- Voluntary switches in the requester process. If a requester calls CHECKSWITCH, the next request comes from the backup process. The process handles for the primary and backup's processes need to swap places.
- File close.

Call FILE_GETRECEIVEINFO_ each time a request is read from $RECEIVE. The process handle and file number must match an opener table entry. If no match is found, reply with error 60 (ZFIL_ERR_WRONGID). This error indicates that the requester had a server with the same name open, that server terminated, and a new server with the same name was started. The OPEN in the requester is still valid, which is why messages might be received even if no preceding OPEN message has been received. You can perform requester error recovery by closing the server file and opening it again, which causes the allocation of a new open control block in the server.

Writing a Requester Program: An Example

The sample requester program given here forms part of a sales-ordering application involving some inventory control and order processing. This part of the application performs three functions:

- Queries the inventory database to find out how much of a given item is on hand.
- Processes an order by updating the inventory database and creating an order record.

- Queries the status of an existing order to find out who placed the order, when the order was placed, and whether the order has been shipped.

User Interface

When the application starts up, it displays the main menu on the terminal as follows:

```
Type 'r' to read a part record
     'p' to process an order
     'q' to query an order
     'x' to exit the program
Choice:
```

When the user selects “r” from the main menu, the application prompts the user for a part number and then displays inventory information about the specified item.

When the user selects “p” from the main menu, the application prompts the user for information to fill out an order request. First it prompts for a part number and displays inventory information. Then it prompts the user to specify the quantity, the name and address of the customer, and the customer’s credit-card number. Once the application has processed the order, it displays an order number on the terminal.

When the user selects “q” from the main menu, the application prompts for an order number. The application responds by displaying information about the order.

When the user selects “x” from the main menu, the process stops and the command-interpreter prompt returns.

Application Overview

The application database is made up of part records and order records. The part records are contained in the inventory file and the order records in the orders file.

The Inventory File

The inventory file contains one record for each item that the store carries. A part record contains the following information about a given part:

- The part number
- A brief description of the item
- The quantity of the item currently on hand
- The unit price of the item
- The name of the supplier
- If an order has been placed with the supplier, the quantity ordered and the expected delivery date

The Orders File

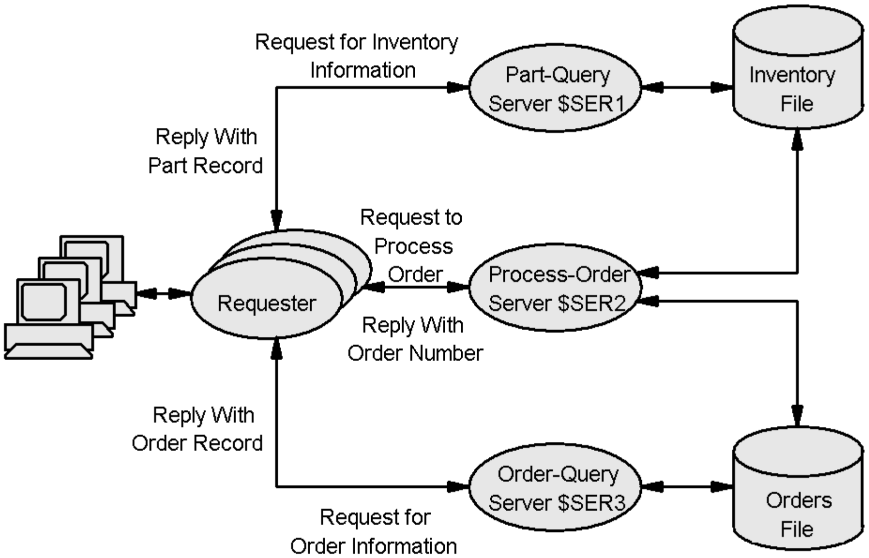
The orders file contains one order record for each item ordered by a customer. The orders file contains the following information:

- The part number of the item ordered
- A brief description of the item ordered
- The quantity ordered
- The name, address, and credit-card number of the customer
- The date when the order was placed
- The date that the order was shipped (if it has been shipped)
- The status of the order, indicating whether the order has been shipped, paid for, and so on

The Role of the Requester in the Application

Figure 21-4 shows the role of the requester in the application.

Figure 21-4. The Requester in the Example Application


Request for Inventory Information
Part-Query Server $SER1
Inventory File
Reply With Part Record
Request to Process Order
Requester
Process-Order Server $SER2
Reply With Order Number
Reply With Order Record
Order-Query Server $SER3
Orders File
Request for Order Information

VST106.VSD

The requester chooses the server to send a request to depending on the function requested by the user:

- If the user requests to query a part record, then the requester obtains the part record by sending a request to the part-query server ($SER1). The $SER1 process obtains the information from the inventory file and returns it to the requester.

- If the user requests to process an order, the requester sends a message to the process-order server ($SER2). This server uses the information it receives to update the inventory level in the inventory file and to create an order record and put it in the orders file.
- If the user wants to query an existing order record, then the requester sends a message to the order-query server ($SER3), which queries the orders file.

See Section 22, Writing a Server Program, for a detailed description of each type of server process.

Enhancements to the Application

Note that for a typical mail-order or telephone-order company, the application is incomplete. The following functions would also usually be required:

- A means for the receiving department to update the inventory file when new shipments of goods are received.
- A means for the billing department to interrogate the orders file to find out to whom to send the invoice.
- A means for the shipping department to examine the orders file to find out to whom to send goods.

Before Running the Application

Before you run the application you need to create the inventory file and the orders file and set up some CLASS MAP DEFINEs required by the application.

You can use the FUP utility to create the inventory and orders files as follows:

```
1> FUP
-SET TYPE K
-SET BLOCK 2048
-SET REC 100
-SET IBLOCK 2048
-SET KEYLEN 10
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 100
    BLOCK  2048
    IBLOCK 2048
    KEYLEN 10
    KEYOFF 0
    MAXEXTENTS 16
-CREATE \SYS.$APPLS.DATA.PARTFILE
CREATED - \SYS.$APPLS.DATA.PARTFILE
-SET TYPE K
```

```
-SET BLOCK 2048
-SET REC 240
-SET IBLOCK 2048
-SET KEYLEN 10
-SHOW
    TYPE K
    EXT ( 1 PAGES, 1 PAGES )
    REC 240
    BLOCK  2048
    IBLOCK 2048
    KEYLEN 10
    KEYOFF 0
    MAXEXTENTS 16
-CREATE \SYS.$APPLS.DATA.ORDERS

CREATED - \SYS.$APPLS.DATA.ORDERS
-EXIT
2>
```

You need to set up a CLASS MAP DEFINE for each of the following files:

- The inventory file
- The orders file
- The program file for the requester
- The program file for each of the servers

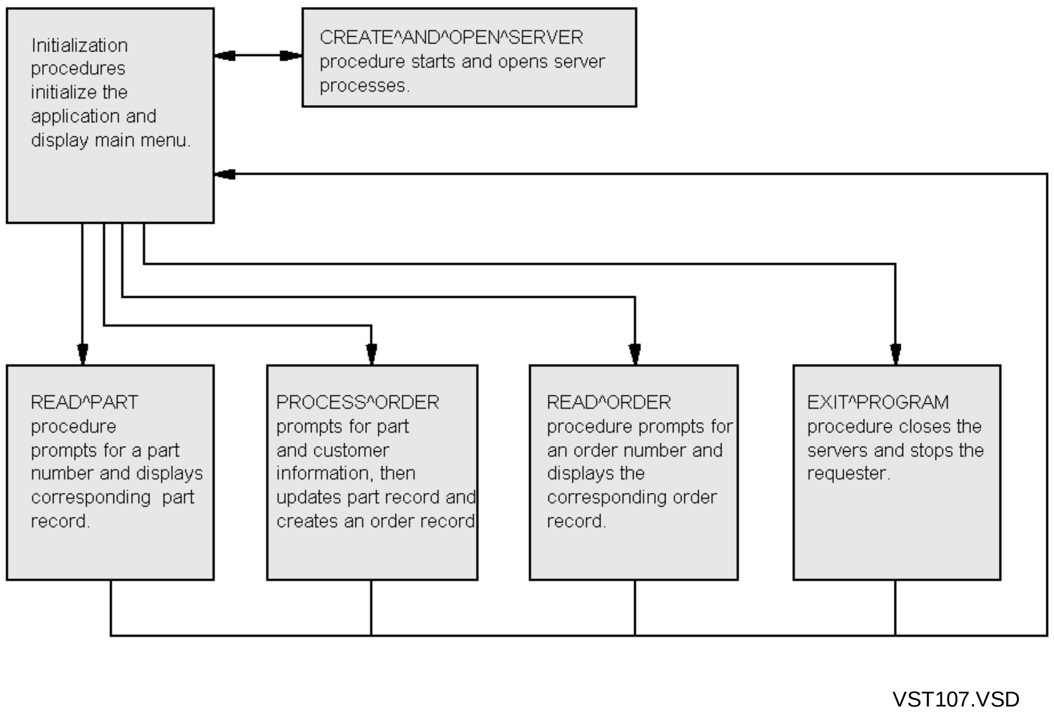
  You can execute an obey file similar to the following to create these DEFINEs:

```
3> O DEFFILE
add define =ser1, class map, file $APPLS.PROGS.zser1
add define =ser2, class map, file $APPLS.PROGS.zser2
add define =ser3, class map, file $APPLS.PROGS.zser3
add define =inv^fname, class map, file $APPLS.DATA.partfile
add define =ord^fname, class map, file $APPLS.DATA.orders
4>
```

Coding the Requester Program

The requester program shown at the end of this section consists of several procedures, as shown in Figure 21-5.

Figure 21-5. Relationship Between Major Procedures in the Requester Program


Initialization procedures initialize the application and display main menu.
CREATE^AND^OPEN^SERVER procedure starts and opens server processes.
READ^PART procedure prompts for a part number and displays corresponding part record.
PROCESS^ORDER prompts for part and customer information, then updates part record and creates an order record
READ^ORDER procedure prompts for an order number and displays the corresponding order record.
EXIT^PROGRAM procedure closes the servers and stops the requester.
VST107.VSD

In addition to the procedures shown in Figure 21-5, the error handling procedures FILE^ERRORS and FILE^ERRORS^NAME provide file-system error handling for most of the other procedures. The WRITE^LINE procedure helps simplify terminal display.

The following paragraphs describe the major procedures in detail.

The Initialization Procedures

The Initialization procedures include the REQUESTER (MAIN), INIT, and GET^COMMAND procedures. These procedures perform two functions: application initialization and main-menu handling.

The INIT procedure performs application initialization. It reads the Startup message, opens the home terminal, and calls the CREATE^AND^OPEN^SERVER procedure once for each server process.

The GET^COMMAND procedure displays the main menu on the terminal, allowing the user to choose the database operation to be performed or to exit the program.

GET^COMMAND passes the result back to the requester procedure which calls the appropriate procedure in response to the user’s selection as follows:

- To read a part record, it calls the READ^PART procedure
- To process an order, it calls the PROCESS^ORDER procedure
- To read an order record, it calls the READ^ORDER procedure
- To stop the requester, it calls the EXIT^PROGRAM procedure

After executing the READ^PART, PROCESS^ORDER, or READ^ORDER procedure, control returns to the REQUESTER procedure, which calls GET^COMMAND again.

The OPEN^SERVER Procedure

The CREATE^AND^OPEN^SERVER procedure works with the OPEN^SERVER procedure to create and open server processes. It is called by the REQUESTER procedure once for each server process. The actions of CREATE^AND^OPEN^SERVER depend on whether the server is already running. The first thing this procedure does is to call the PROCESS_GETPAIRINFO_ system procedure to see whether the server process already exists.

If the server already exists, then CREATE^AND^OPEN^SERVER calls OPEN^SERVER to try to open it. The attempt to open succeeds unless the server rejects the attempt because its opener table is full. If the open is rejected, then the OPEN^SERVER procedure prompts the user to try again to open the server; you can either keep trying or choose to quit and exit the program. If the procedure succeeds in opening the server, control returns to the REQUESTER procedure.

If the server does not exist, then CREATE^AND^OPEN^SERVER calls the PROCESS_CREATE_ system procedure to create it. Following the normal startup protocol, the procedure then calls OPEN^SERVER to open the server, sends it a Startup message, closes the server, and then opens it again. If either open fails, then the OPEN^SERVER procedure again allows the user to retry the operation.

The READ^PART Procedure

The READ^PART procedure interacts with the part-query server ($SER1) to read a part record given a part number. It is called from the REQUESTER procedure when the user types “r” in response to the main-menu prompt.

On input, the terminal-interface phase of this procedure prompts the user for a part number. The procedure expects a 10-digit number in reply.

The field-validation phase checks that the part number is 10 digits long and consists entirely of numeric characters. If either of these conditions is not met, then the procedure prompts the user to enter another part number.

The input is already in the form that the server expects it (a 10-digit number string), therefore no data-mapping phase is required on input.

The application-control phase sends the 10-digit number string to the $SER1 server process and waits for the response.

For an existing part number, the server returns a data structure containing the part record and the READ^PART procedure displays the information on the terminal. Date information contained in the returned data structure is in the form of a 48-bit numeric timestamp. The READ^PART procedure converts this information first into a 16-bit integer representing the Gregorian date and time and then converts the numbers into ASCII characters for output.

If the part number does not exist, the server returns an error and the READ^PART procedure prompts the user for another part number.

The PROCESS^ORDER Procedure

The PROCESS^ORDER procedure communicates with the process-order server ($SER2) to process a customer order. It is called from the REQUESTER procedure when the user types “p” in response to the main-menu prompt.

This procedure prompts the terminal operator to enter the part number of the item to be ordered, the quantity, and the customer’s name, address, and credit-card number.

This procedure first calls the READ^PART procedure to prompt for the part number and provide the operator with inventory information to see whether the store can satisfy the order.

The field-validation phase enforces the following:

- The quantity requested must be numeric.
- The customer’s first name and last name must be alphabetic and have from 1 through 20 characters. The middle initial should be a single character or omitted.
- The customer’s street address must contain up to 48 alphabetic and numeric characters.
- The city name can be up to 24 characters long, all of which must be alphabetic.
- The zip code (for the purpose of this example) must consist of seven characters: the first two characters must be alphabetic, and the remaining five must be numeric.
- The customer’s credit-card number must be 16 numeric characters.

The PROCESS^ORDER procedure prompts the user to reenter any part of the above information that does not meet the stated requirements.

The data-mapping phase involves converting the ASCII input for the quantity into a numeric value and then packing all input into a data structure to send to the server.

Application control involves selecting the server to send the data structure to. In this case, the server is the $SER2 process. The PROCESS^ORDER procedure then waits for the reply.

If the server process successfully processes the order, then the reply record contains the new stock level on hand and an order record number for the newly created order. This number is 28 digits long and is made up of a timestamp and the part number. The PROCESS^ORDER procedure displays the order number on the terminal.

If the reply structure returns a negative quantity on hand, then the PROCESS^ORDER procedure informs the user that the order cannot be satisfied.

If the server process cannot process the order for any reason other than inadequate inventory, then an error condition is returned.

The READ^ORDER Procedure

The READ^ORDER procedure interacts with the order-query server ($SER3) to read an order record given an order number. It is called from the REQUESTER procedure when the user types “q” in response to the main-menu prompt.

On input, the terminal-interface phase of this procedure prompts the user for an order number. The procedure expects a 28-digit number in reply.

The field-validation phase checks that the order number is 28 digits long and consists entirely of numeric characters. If either of these conditions is not met, then the procedure prompts the user to enter another order number.

The input is already in the form that the server expects (a 28-digit number string), therefore no data-mapping phase is required on input.

The application-control phase sends the 28-digit number string to the $SER3 process and waits for the response.

For an existing order number, the server returns a data structure containing the order record and the READ^ORDER procedure displays the information on the terminal. As with the READ^PART procedure, date information is converted for output.

If the order number does not exist, the server returns an error and the READ^ORDER procedure prompts the user for another order number.

The EXIT^PROGRAM Procedure

The EXIT^PROGRAM procedure simply calls the FILE_CLOSE_ procedure for each server (allowing each server to delete an entry from its opener table) and then calls the PROCESS_STOP_ procedure to stop the requester.

The ERROR^HANDLER Procedure

The ERROR^HANDLER procedure gets called from several procedures in the requester to handle file-system errors. This procedure interfaces with the ERROR program to print a brief description of the file-system error. The interface with the ERROR process is described in detail in Section 20, Interfacing With the ERROR Program.

The Code for the Sample Requester Program

The rest of this section lists the code for the sample requester program.

```
?INSPECT, SYMBOLS, NOCODE
!NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

!---------------
!Literals:
!---------------

LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME;
LITERAL BUFSIZE = 512;

!----------------
!Data structures
!----------------

!Startup message data structure:

STRUCT .START^UP^MESSAGE;   !Startup message to send to
BEGIN                       ! server
   INT MSG^CODE;            !-1 for Start-Up message
   STRUCT DEFAULT;          !default file name
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;           !INFILE name
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILENAME[0:3];
   END;
   STRUCT OUTFILE;          !OUTFILE name
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILENAME[0:3];
   END;
   STRING PARAM[0:529];     !parameter string
END;
INT  MESSAGE^LEN;           !length of Startup message
```

```
!Message to send to $SER1 to request inventory information
!about a specified item:

STRUCT PART^REQUEST;
BEGIN
   STRING PART^NUMBER[0:9]; !10-digit part number
END;


!Message to send to $SER2 to process an order:

STRUCT .ORDER^REQUEST;
BEGIN
   STRUCT NAME;             !name of customer
   BEGIN
      STRING LAST[0:19];
      STRING FIRST[0:19];
      STRING INITIAL[0:1];
   END;
   STRING ADDRESS[0:47];    !address of customer
   STRING CITY[0:23];       !city
   STRING ZIP[0:7];         !customer's zip code
   STRING CCN[0:15];        !customer's credit card number
   STRING PART^NUMBER[0:9]; !part number for part ordered
   STRING PART^DESC[0:47];  !description of part ordered
   INT    QTY^ORDERED;      !quantity ordered
END;
```

```
!Data structure send to $SER3 to request information about a
!specified order:

STRUCT .ORDER^QUERY;
BEGIN
   STRING ORDER^NUMBER[0:27];
END;


!Message returned by $SER1.  It contains a part record:

STRUCT .PART^REC;
BEGIN
   STRING PART^NUMBER[0:9]; !10-digit part number
   STRING PART^DESC[0:47];  !part description
   STRING SUPPLIER[0:23];   !name of part supplier
   INT    QUANTITY^ON^HAND; !quantity of parts on hand
   INT    UNIT^PRICE;       !price of part
   INT    ORDER^PLACED[0:2];!date order placed with supplier
   INT    SHIPMENT^DUE[0:2];!date order due from supplier
   INT    QUANTITY^ORDERED; !quantity of part on order from
END;                        ! supplier


!Message returned by $SER2.  After processing
!an order request, $SER2 returns this data structure
!containing the stock balance of the item ordered,
!and the order number:

STRUCT .ORDER^REPLY;
BEGIN
   INT    QUANTITY^ON^HAND; !quantity after order satisfied
   STRING ORDER^NUMBER[0:27];!28-digit order number
END;
```

```
!Message returned by $SER3.  It contains the order record
!the corresponds to the order number sent in the request
!to $SER3:

STRUCT .ORDER^REC;
BEGIN
   STRING ORDER^NUMBER[0:27];!order number
   STRUCT NAME;             !name of customer
   BEGIN
      STRING LAST[0:19];
      STRING FIRST[0:19];
      STRING INITIAL[0:1];
   END;
   STRING ADDRESS[0:47];    !address of customer
   STRING CITY[0:23];       !city
   STRING ZIP[0:7];         !customer's zip code
   STRING CCN[0:15];        !customer's credit card number
   STRING PART^NUMBER[0:9]; !part number for part ordered
   STRING PART^DESC[0:47];  !description of part ordered
   INT    QTY^ORDERED;      !quantity ordered
   INT    DATE^ORDERED[0:2];!date when customer placed order
   INT    DATE^SHIPPED[0:2];!date when shipped to customer
   STRING SHIPPING^STATUS[0:1];!status of order; shipped,
END;                        ! not shipped...


!-----------------------
!Other global variables:
!-----------------------

STRING  PART^NUMBER[0:9];   !10-digit part number
INT     SERV1^NUM;          !file number for $SER1
INT     SERV2^NUM;          !file number for $SER2
INT     SERV3^NUM;          !file number for $SER3
INT     DATE^AND^TIME[0:6]; !converted 48-bit time stamp

INT     TERM^NUM;           !file number for home terminal
STRING .SBUFFER[0:BUFSIZE]; !I/O buffer
STRING .S^PTR;              !string pointer

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_OPEN_,FILE_CLOSE_,
?                       PROCESS_CREATE_,PROCESS_GETPAIRINFO_,
?                       PROCESS_STOP_,FILE_GETINFO_,CONTIME,
?
NUMIN,INITIALIZER,OLDFILENAME_TO_FILENAME_,
?                       DNUMOUT,WRITEX,WRITEREADX)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N) =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE PRINT^LINE =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

   DEFINE PRINT^STR(S) =  BEGIN  START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;   END   #;
```

```
!------------------------------------------------------------
! Procedure for displaying file system error numbers on the
! terminal.  The parameters are the file name and its length
! and the error number.  This procedure is used when the
! file is not open, so there is no file number for it.
! FILE^ERRORS is used when the file is open.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING         .FNAME;
INT             LEN;
INT             ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN );

   CALL WRITEX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER);

   START^LINE;
   PUT^STR("occurred in requester program ");
   CALL WRITEX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure for displaying file system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number.
! FILE^ERRORS^NAME is called to display the information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT   FNUM;
BEGIN
   INT      ERROR;
   STRING  .FNAME[0:MAXFLEN - 1];
   INT      FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);
END;


!-------------------------------------------------------------
! Procedure to prompt the user for the next function to be
! performed:
!
! "r" to read a part record
! "p" to process an order
! "q" to query an order
! "x" to exit the program
!
! The selection made is returned as a result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT    COUNT^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type 'r' to read a part record, ");
   PRINT^STR("     'p' to process an order, ");
   PRINT^STR("     'q' to query an order, ");
   PRINT^STR("     'x' to exit.  ");

   SBUFFER ':=' "Choice: " -> @S^PTR;
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);

   SBUFFER[COUNT^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!-------------------------------------------------------------
! Procedure to read a part record from the $SER1 process.
! This procedure prompts the user for a part number, which it
! checks for validity before sending it to the $SER1 server
! process.  The $SER1 process returns the corresponding part
! record (if there is one) from the inventory file and then
! this procedure prints the record on the terminal.
!
! This procedure is called from the main procedure when the
! user selects "r" from the main menu.  It is also called
! from the PROCESS^ORDER procedure to display stock-level
! information before processing the order.
!-------------------------------------------------------------

PROC READ^PART;
BEGIN
   INT COUNT^READ;
   INT ERROR;
   INT I;

!  Repeat until a valid part number entered:

REPEAT:

!  Request a part number from the terminal user:

   START^LINE;
   PUT^STR("Enter Part Number:  ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);

!  Check that part number contains 10 characters. Request
!  another part number if not:

   IF COUNT^READ <> 10 THEN
   BEGIN

   !  Print diagnostic for failed test:

      PRINT^LINE;
      PRINT^STR("Part Number Must Contain 10 Characters");
      PRINT^STR("Please type another part number ");
      PRINT^BLANK;
      GOTO REPEAT;
   END;
```

```
!  Check if part number all numeric.  Request another part
!  number if not:

   I := 0;
   WHILE I < 10 DO
   BEGIN
      IF SBUFFER[I] < "0" OR SBUFFER[I] > "9" THEN
      BEGIN

      !  Print diagnostic for failed test:

         PRINT^BLANK;
         PRINT^STR("Part Number Must Be Numeric ");
         PRINT^STR("Please type another part number ");
         PRINT^BLANK;
         GOTO REPEAT;
      END;
      I := I + 1;
   END;

!  Send part number to server:

   CALL WRITEREADX(SERV1^NUM,SBUFFER,$LEN(PART^REQUEST),
                   BUFSIZE,COUNT^READ);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERV1^NUM,ERROR);
      CASE ERROR OF
      BEGIN

      !  If server could not find a record with supplied key
      !  (part number) print message and request another part
      !  number:

         11 -> BEGIN

         !  Print diagnostic:

            PRINT^BLANK;
            PRINT^STR("No Such Part Number ");
            PRINT^STR("Please Type Another Part Number ");
            PRINT^BLANK;
            GOTO REPEAT;
         END;

         OTHERWISE -> BEGIN

         !  Other error, call FILE^ERRORS to display error
         !  and exit the process:

            CALL FILE^ERRORS(TERM^NUM);
         END;
      END;
   END;
```

```
!  Print two blank lines on the terminal:

   PRINT^BLANK;
   PRINT^BLANK;

!  Print returned information on the terminal:

   PART^REC ':=' SBUFFER FOR ($LEN(PART^REC));
   PRINT^STR("INVENTORY PROFILE:  ");

!  Print the part number:

   PRINT^STR("Part number:        " &
             PART^REC.PART^NUMBER FOR 10);

!  Print the part description:

   PRINT^STR("Part description:   " &
              PART^REC.PART^DESC FOR 48);

!  Print the quantity on hand:

   START^LINE;
   PUT^STR("Quantity on Hand:   ");
   PUT^INT(PART^REC.QUANTITY^ON^HAND);
   PRINT^LINE;

!  Print the unit price:

   START^LINE;
   PUT^STR("Unit Price:         $");
   PUT^INT(PART^REC.UNIT^PRICE);
   PRINT^LINE;

   PRINT^BLANK;

!  Print out any reorder information:

   PRINT^STR("REORDER INFORMATION:");
```

```
!  Print the supplier's name:

   PRINT^STR("Supplier Name:      " &
             PART^REC.SUPPLIER FOR 24);

!  Print date when last order placed with supplier:

   CALL CONTIME(DATE^AND^TIME,PART^REC.ORDER^PLACED[0],
        PART^REC.ORDER^PLACED[1],PART^REC.ORDER^PLACED[2]);
   START^LINE;
   PUT^STR("Order Placed:       ");
   PUT^INT(DATE^AND^TIME[1]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[2]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[0]);
   PRINT^LINE;

!  Print date when next shipment is due from the supplier:

   START^LINE;
   PUT^STR("Shipment Due:       ");
   CALL CONTIME(DATE^AND^TIME,PART^REC.SHIPMENT^DUE[0],
        PART^REC.SHIPMENT^DUE[1],PART^REC.SHIPMENT^DUE[2]);
   PUT^INT(DATE^AND^TIME[1]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[2]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[0]);
   PRINT^LINE;

!  Print quantity ordered from supplier:

   START^LINE;
   PUT^STR("Quantity Ordered:   ");
   PUT^INT(PART^REC.QUANTITY^ORDERED);
   PRINT^LINE;


   PRINT^BLANK;
   PRINT^BLANK;
END;
```

```
!-------------------------------------------------------------
! Procedure to process an order.  The procedure puts together
! a request for an order from information entered in response
! to a series of prompts.  After extensive checking, sends
! the request to the $SER2 server process which does the
! following:
!
!   -- Updates the corresponding PART^REC
!   -- Places an order record in the orders file
!   -- Returns a reply record to this procedure
!
! The reply record contains an indication of the new stock
! level,and a 28-digit order number made up from a converted
! time stamp and the part number.  This procedure prints the
! order number on the terminal.
!-------------------------------------------------------------

PROC PROCESS^ORDER;
BEGIN
   INT    COUNT^READ;
   INT    BASE;
   INT    STATUS;
   INT    I;

!  Blank the order-request message structure:

   ORDER^REQUEST ':=' ($LEN(ORDER^REQUEST) / 2) * ["  "];


!----------------------------
!  Prompt for and process the
!  part number.
!----------------------------

!  Call the READ^PART procedure to prompt for the part number
!  and find out if enough stock is on hand:

   CALL READ^PART;
```

```
!----------------------------
!  Prompt for and process the
!  quantity requested
!----------------------------

!  Repeat until valid quantity entered:

REPEAT^QTY:

!  Prompt for the quantity required:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Quantity: ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,COUNT^READ);

!  Check that input is numeric:

   I := 0;
   WHILE I < COUNT^READ DO
   BEGIN
      IF SBUFFER[I] < "0" OR SBUFFER[I] > "9" THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("Quantity must be numeric");
         GOTO REPEAT^QTY;
      END;
      I := I + 1;
   END;

!  Convert input from a numeric string to a number, and place
!  it in the ORDER^REQUEST message:

   BASE := 10;
   CALL NUMIN(SBUFFER[0],ORDER^REQUEST.QTY^ORDERED,
              BASE,STATUS);

!  If status indicates an error, print diagnostic and prompt
!  again for the quantity:

   IF STATUS <> 0 THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Invalid input ");
      PRINT^STR("Please enter a valid number ");
      GOTO REPEAT^QTY;
   END;
```

```
!  Check that there is enough stock on hand to satisfy the
!  order:

   IF ORDER^REQUEST.QTY^ORDERED > PART^REC.QUANTITY^ON^HAND
   THEN
   BEGIN
      PRINT^BLANK;
      START^LINE;
      PUT^STR("Current Stock on Hand is Only ");
      PUT^INT(PART^REC.QUANTITY^ON^HAND);
      PRINT^LINE;
      RETURN;
   END;


!----------------------------
!  Prompt for and process the
!  customer's last name
!----------------------------

!  Repeat until valid last name entered:

REPEAT^LASTNAME:

!  Prompt user for last name:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Customer's Last Name: ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,COUNT^READ);


!  If name is greater than 20 characters or less than one
!  character, prompt user to enter a name of valid length:

   IF COUNT^READ > 20 OR COUNT^READ < 1 THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Last Name Must Be 1 to 20 Characters");
      PRINT^STR("Please Enter a Valid Name ");
      GOTO REPEAT^LASTNAME;
   END;
```

```
!  If name contains nonalphabetic characters, prompt user to
!  reenter name:

   I := 0;
   WHILE I < COUNT^READ DO
   BEGIN
      IF SBUFFER[I] < "A" OR SBUFFER[I] > "z" OR
      (SBUFFER[I] > "Z" AND SBUFFER[I] < "a") THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("Name Must Be Alphabetic ");
         PRINT^STR("Please Enter an Alphabetic Last Name");
         GOTO REPEAT^LASTNAME;
      END;
      I := I + 1;
   END;

!  Put last name in order-request message:
   ORDER^REQUEST.NAME.LAST ':=' SBUFFER FOR COUNT^READ;


!----------------------------
!  Prompt for and process the
!  customer's first name
!----------------------------

!  Repeat until valid first name entered:

REPEAT^FIRSTNAME:

!  Prompt user for first name:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Customer's First Name:  ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);

!  If name is greater than 20 characters or less than one
!  character, prompt the user to enter a name of valid
!  length:

   IF COUNT^READ > 20 OR COUNT^READ < 1 THEN
   BEGIN

      PRINT^BLANK;
      PRINT^STR("First Name Must Be 1 to 20 Characters ");
      PRINT^STR("Please Enter a Valid name ");
      GOTO REPEAT^FIRSTNAME;
   END;
```

```
!  If first name contains nonalphabetic characters, prompt
!  user to reenter name:

   I := 0;
   WHILE I < COUNT^READ DO
   BEGIN
      IF SBUFFER[I] < "A" OR SBUFFER[I] > "z" OR
      (SBUFFER[I] > "Z" AND SBUFFER[I] < "a") THEN
      BEGIN
         PRINT^STR("Name Must Be Alphabetic ");
         PRINT^STR("Please Enter an Alphabetic First Name ");
         GOTO REPEAT^FIRSTNAME;
      END;
      I := I + 1;
   END;

!  Put first name in order-request message:

   ORDER^REQUEST.NAME.FIRST ':=' SBUFFER[0] FOR COUNT^READ;


!-----------------------------
!  Prompt for and process the
!  customer's middle initial
!-----------------------------

!  Repeat until valid middle initial entered:

REPEAT^INITIAL:

!  Prompt user for middle initial:

      PRINT^BLANK;
      START^LINE;
      PUT^STR("Enter Customer's Middle Initial:  ");
      CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);

!  If middle initial is greater than 1 character, prompt
!  user to issue a single character:

   IF COUNT^READ > 1 THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Middle Initial Must Be 1 or 0 Characters");
      PRINT^STR("Please enter a single character ");
      GOTO REPEAT^INITIAL;
   END;
```

```
!  If middle initial is nonalphabetic, and not blank, prompt
!  user to reenter middle initial:

   IF COUNT^READ = 1 THEN
   BEGIN
      IF SBUFFER[0] < "A" OR SBUFFER[0] > "z" OR
      (SBUFFER[0] > "Z" AND SBUFFER[0] < "a") AND
      SBUFFER[0] <> " " THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("Middle Initial Must Be Alphabetic ");
         PRINT^STR("Please Enter an Alphabetic Character");
         GOTO REPEAT^INITIAL;
      END;
   END;

!  Put middle initial in order-request message:

   ORDER^REQUEST.NAME.INITIAL ':=' SBUFFER[0] FOR COUNT^READ;


!---------------------------
! Prompt for and process the
! customer's street address
!---------------------------

!  Repeat until valid address entered:

REPEAT^ADDR:

!  Prompt user for address:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Customer's Street Address:  ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);

!  If address is greater than 48 characters or less than one
!  character, prompt user to enter a valid address:

   IF COUNT^READ > 48 OR COUNT^READ < 1THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Address Must Be 1 to 48 Characters  ");
      PRINT^STR("Please Enter a Valid Address");
      GOTO REPEAT^ADDR;
   END;
```

```
!  Put street address in order-request message:

   ORDER^REQUEST.ADDRESS ':=' SBUFFER[0] FOR COUNT^READ;

!-----------------------------
!  Prompt for and process the
!  customer's city name
!-----------------------------

!  Repeat until valid city name entered:

REPEAT^CITY:

!  Prompt user for city name:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter City Name:  ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,
                   @S^PTR '-' @SBUFFER,BUFSIZE,
                   COUNT^READ);

!  If name is greater than 24 characters, prompt user to
!  issue a shorter city name:

   IF COUNT^READ > 24 OR COUNT^READ < 1 THEN
   BEGIN

      PRINT^BLANK;
      PRINT^STR("City Name Must Be 1 To 24 Characters");
      PRINT^STR("Please Enter a Valid city name");
      GOTO REPEAT^CITY;
   END;

!  If city name contains nonalphabetic characters, prompt
!  user to reenter city name:

   I := 0;
   WHILE I < COUNT^READ DO
   BEGIN
      IF SBUFFER[I] < "A" OR SBUFFER[I] > "z" OR
      (SBUFFER[I] > "Z" AND SBUFFER[I] < "a") THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("City Name Must Be Alphabetic");
         PRINT^STR("Please Enter an Alphabetic City Name");
         GOTO REPEAT^CITY;
      END;
      I := I + 1;
   END;

!  Put city name in order-request message:
   ORDER^REQUEST.CITY ':=' SBUFFER[0] FOR COUNT^READ;
```

```
!----------------------------
!  Prompt for and process the
!  customer's zip code
!----------------------------

!  Repeat until valid zip code entered:

REPEAT^ZIP:

!  Prompt user for zip code:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Zip Code: ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);

!  If zip code does not have exactly 7 characters, prompt
!  user to issue another zip code:

   IF COUNT^READ <> 7 THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Zip Code Must Have Exactly 7 Characters ");
      PRINT^STR("Please Enter a Valid Zip Code ");
      GOTO REPEAT^ZIP;
   END;

!  If either of the first two characters of the zip code is
!  nonalphabetic, reenter the zip code:

   I := 0;
   WHILE I < 2 DO
   BEGIN
      IF SBUFFER[I] < "A" OR SBUFFER[I] > "z" OR
      (SBUFFER[I] > "Z" AND SBUFFER[I] < "a") THEN
      BEGIN
         PRINT^STR("First Two Characters Must Be ” &
                   "Alphabetic ");
         PRINT^STR("Please Enter an Alphabetic Characters ");
         GOTO REPEAT^ZIP;
      END;
      I := I + 1;
   END;
```

```
!  If any of the last five characters of the zip code is
!  nonnumeric, reenter the zip code:

   I := 2;
   WHILE I < 7 DO
   BEGIN
      IF SBUFFER[I] < "0" OR SBUFFER[I] > "9" THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("Last Five Characters Must Be Numeric");
         PRINT^STR("Please Enter Numeric Characters ");
         GOTO REPEAT^ZIP;
      END;
      I := I + 1;
   END;

!  Put zip code in order-request message:

   ORDER^REQUEST.ZIP ':=' SBUFFER[0] FOR COUNT^READ;


!-------------------------------
!  Prompt for and process the
!  customer's credit card number
!-------------------------------

!  Repeat until valid credit-card number entered:

REPEAT^CCN:

!  Prompt user for credit-card number:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Enter Customer's Credit-Card Number: ");
   CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,COUNT^READ);

!  If credit-card number not exactly 16 characters,
!  prompt user to enter a valid credit-card number:

   IF COUNT^READ <> 16 THEN
   BEGIN

      PRINT^BLANK;
      PRINT^STR("Credit-Card Number Must be 16 Characters ");
      PRINT^STR("Please Enter a Valid Credit-Card Number ");
   END;
```

```
!  Check that credit-card number is all numeric:

   I := 0;
   WHILE I < 16 DO
   BEGIN
      IF SBUFFER[I] < "0" OR SBUFFER[I] > "9" THEN
      BEGIN
         PRINT^BLANK;
         PRINT^STR("Credit Card Number Must Be All Numeric");
         PRINT^STR("Please Enter Valid Credit-Card Number");
         GOTO REPEAT^CCN;
      END;
      I := I + 1;
   END;

!  Put credit-card number in order-request message:
   ORDER^REQUEST.CCN ':=' SBUFFER[0] FOR COUNT^READ;


!---------------------------
!  Prepare part of order
!  request that does not
!  need user input
!---------------------------

!  Copy part number from PART^REC:

   ORDER^REQUEST.PART^NUMBER ':=' PART^REC.PART^NUMBER
                                  FOR 10;

!  Copy part description from PART^REC:

   ORDER^REQUEST.PART^DESC ':=' PART^REC.PART^DESC FOR 48;


!---------------------------
!  Process the request
!---------------------------

!  Put request record into I/O buffer:

   SBUFFER ':=' ORDER^REQUEST for ($LEN(ORDER^REQUEST) / 2);

!  Send request to server:

   CALL WRITEREADX(SERV2^NUM,SBUFFER,$LEN(ORDER^REQUEST),
                   BUFSIZE,COUNT^READ);
   IF <> THEN CALL FILE^ERRORS(SERV2^NUM);
```

```
!  Copy reply from server into ORDER^REPLY structure:

   ORDER^REPLY ':=' SBUFFER FOR $LEN(ORDER^REPLY);

!  If stock depleted since checking, inform user and return:

   IF ORDER^REPLY.QUANTITY^ON^HAND < 0 THEN
   BEGIN
      PRINT^BLANK;
      PRINT^STR("Insufficient Stock for this Order ");
      RETURN;
   END;

!  Prepare the order number for printing in blocks of
!  6 characters separated by spaces:

   PRINT^BLANK;
   START^LINE;
   PUT^STR("Order Number is:  ");
   PUT^STR(ORDER^REPLY.ORDER^NUMBER[0] FOR 6);
   PUT^STR(" ");
   PUT^STR(ORDER^REPLY.ORDER^NUMBER[6] FOR 6);
   PUT^STR(" ");
   PUT^STR(ORDER^REPLY.ORDER^NUMBER[12] FOR 6);
   PUT^STR(" ");
   PUT^STR(ORDER^REPLY.ORDER^NUMBER[18] FOR 6);
   PUT^STR(" ");
   PUT^STR(ORDER^REPLY.ORDER^NUMBER[24] FOR 4);

!  Print order number on the terminal:

   PRINT^LINE;
END;
```

```
!------------------------------------------------------------
! Procedure to read an order record from the $SER3 process.
! This procedure prompts the user for an order number, which
! it checks for validity before sending it to the $SER3
! server process.  The $SER3 process returns the order record
! (if there is one) from the corresponding the order file and
! then this procedure prints the record on the terminal.
!
! This procedure is called from the main procedure when the
! user selects "q" from the main menu.
!------------------------------------------------------------

PROC READ^ORDER;
BEGIN

   INT    COUNT^READ;
   INT    ERROR;
   INT    I;

   !-----------------------------
   !  Prompt for and validate an
   !  order record number.
   !-----------------------------

   REPEAT:

   !  Request an order number from the terminal user:

      START^LINE;
      PUT^STR("Enter Order Number: ");
      CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,COUNT^READ);
      IF <> THEN CALL FILE^ERRORS(TERM^NUM);

   !  Check that order number contains 28 characters. Request
   !  another order number if not:

      IF COUNT^READ <> 28 THEN
      BEGIN

      !  Print diagnostic for failed test:

         PRINT^BLANK;
         PRINT^STR ("Order Number Must Have 28 Characters");
         PRINT^STR ("Please Type Another Order Number");
         PRINT^BLANK;
         GOTO REPEAT;
      END;
```

```
!  Check whether order number all numeric.  Request
!  another order number if not:

   I := 0;
   WHILE I < 28 DO
   BEGIN
      IF SBUFFER[I] < "0" OR SBUFFER[I] > "9" THEN
      BEGIN

      !  Print diagnostic for failed test:

         PRINT^BLANK;
         PRINT^STR("Order Number Must Be Numeric");
         PRINT^STR("Please Type Another Order Number");
         PRINT^BLANK;
         GOTO REPEAT;
      END;
      I := I + 1;
   END;

!------------------------------
!  Get the order record from
!  the server.
!------------------------------

!  Send order number to server:

   CALL WRITEREADX(SERV3^NUM,SBUFFER,$LEN(ORDER^QUERY),
                   BUFSIZE);
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(SERV3^NUM,ERROR);
      CASE ERROR OF
      BEGIN

      !  If server could not find a record with supplied
      !  key (order number), print message and request
      !  another order number:

      11 -> BEGIN

         !  Print diagnostic:

            PRINT^BLANK;
            PRINT^STR("No Such Order Number");
            PRINT^STR("Please Type Another Order Number");
            PRINT^BLANK;
         END;

         !  Other error, call FILE^ERRORS to display error
         !  and exit the process:

         OTHERWISE -> CALL FILE^ERRORS(SERV3^NUM);
   END;
END;
```

```
!---------------------------------
!  Print the contents of the order
!  record on the terminal.
!---------------------------------

   PRINT^BLANK;
   PRINT^BLANK;

!  Copy returned information into ORDER^REC structure:

   ORDER^REC ':=' SBUFFER FOR ($LEN(ORDER^REC));
   PRINT^STR("ORDER RECORD INFORMATION");

!  Print the order number in groups of 6 characters,
!  separated by spaces:

   PRINT^STR("Order Number:       " &
             ORDER^REC.ORDER^NUMBER[0] FOR 6 & " " &
             ORDER^REC.ORDER^NUMBER[6] FOR 6 & " " &
             ORDER^REC.ORDER^NUMBER[12] FOR 6 & " " &
             ORDER^REC.ORDER^NUMBER[18] FOR 6 & " " &
             ORDER^REC.ORDER^NUMBER[24] FOR 4);

!  Print the customer's name:

   PRINT^BLANK;
   PRINT^STR("Customer Name:      " &
             ORDER^REC.NAME.LAST FOR 20 &
             ORDER^REC.NAME.FIRST FOR 20 &
             ORDER^REC.NAME.INITIAL FOR 1);

!  Print the customer's address:
   PRINT^STR("Customer's Address: " &
             ORDER^REC.ADDRESS FOR 48);

!  Print the customer's city:

   PRINT^STR("                    " &
             ORDER^REC.CITY FOR 24);

!  Print the customer's zip code:

   PRINT^STR("                    " &
             ORDER^REC.ZIP[0] FOR 2 & " " &
             ORDER^REC.ZIP[2] FOR 5);

!  Print credit-card number in groups of 4 characters,
!  separated by spaces:

   PRINT^BLANK;
   PRINT^STR("Credit-Card Number: " &
                   ORDER^REC.CCN[0] FOR 4 &
             " " & ORDER^REC.CCN[4] FOR 4 &
             " " & ORDER^REC.CCN[8] FOR 4 &
             " " & ORDER^REC.CCN[12] FOR 4);
```

```
!  Print the part number:

   PRINT^BLANK;
   PRINT^STR("Part Number:        " &
             ORDER^REC.PART^NUMBER FOR 10);

!  Print the part description:

   PRINT^STR("Part description:   " &
             ORDER^REC.PART^DESC FOR 48);

!  Print the quantity ordered:

   START^LINE;
   PUT^STR("Quantity Ordered:   ");
   PUT^INT(ORDER^REC.QTY^ORDERED);
   PRINT^LINE;

!  Print date ordered:

   CALL CONTIME(DATE^AND^TIME,ORDER^REC.DATE^ORDERED[0],
                ORDER^REC.DATE^ORDERED[1],
                ORDER^REC.DATE^ORDERED[2]);
   START^LINE;
   PUT^STR("Date Ordered:       ");
   PUT^INT(DATE^AND^TIME[1]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[2]);
   PUT^STR("-");
   PUT^INT(DATE^AND^TIME[0]);
   PRINT^LINE;
```

```
!  Print date shipped to customer:

   START^LINE;
   PUT^STR("Date Shipped:       ");
   IF ORDER^REC.DATE^SHIPPED = 0 THEN
      PUT^STR("Order Not Yet Shipped ")
   ELSE
   BEGIN
      CALL CONTIME(DATE^AND^TIME,PART^REC.SHIPMENT^DUE[0],
                   PART^REC.SHIPMENT^DUE[1],
                   PART^REC.SHIPMENT^DUE[2]);
      START^LINE;
      PUT^INT(DATE^AND^TIME[1]);
      PUT^STR("-");
      PUT^INT(DATE^AND^TIME[2]);
      PUT^STR("-");
      PUT^INT(DATE^AND^TIME[0]);
   END;
   PRINT^LINE;

!  Print shipping status:

   PRINT^STR("Shipping Status:    " &
             ORDER^REC.SHIPPING^STATUS FOR 2);

   PRINT^BLANK;
   PRINT^BLANK;
END;


!------------------------------------------------------------
! Procedure closes all servers opened by this process and
! then exits.  This procedure is called from the main
! procedure when the user selects "x" from the main menu.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL FILE_CLOSE_(SERV1^NUM);
   CALL FILE_CLOSE_(SERV2^NUM);
   CALL FILE_CLOSE_(SERV3^NUM);
   CALL PROCESS_STOP_;
END;
```

```
!-------------------------------------------------------------
! Procedure opens a server process.  Prompts the user to try
! again if the open fails.
!-------------------------------------------------------------

PROC OPEN^SERVER(PROCESS^NAME,PROCESS^NAMELEN,SERVER^NUM);
STRING .PROCESS^NAME;
INT     PROCESS^NAMELEN;
INT    .SERVER^NUM;

BEGIN
   INT     ERROR;

TRY^AGAIN:

   ERROR := FILE_OPEN_(PROCESS^NAME:PROCESS^NAMELEN,
                       SERVER^NUM);
   IF ERROR <> 0 THEN
   BEGIN
      PRINT^STR("Could not open server");
      SBUFFER ':=' "Do you wish to try again? (y/n): "
                 -> @S^PTR;
      CALL WRITEREADX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE);
      IF (SBUFFER[0] = "n") OR (SBUFFER[0] = "N") THEN
         CALL PROCESS_STOP_
         ELSE GOTO TRY^AGAIN;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure handles creating and opening servers.  If the
! server already exists it calls OPEN^SERVER to open it.  If
! it does not exist, it creates the server and sends it the
! standard process initialization sequence.
!------------------------------------------------------------

PROC CREATE^AND^OPEN^SERVER(SERVER^NUM,SERVER^OBJECT^NAME,
                            OBJFILE^NAMELEN,PROCESS^NAME,
                            PROCESS^NAMELEN);

INT    .SERVER^NUM;           !file number of server process
STRING .SERVER^OBJECT^NAME;   !name of server object file
INT     OBJFILE^NAMELEN;
STRING .PROCESS^NAME;         !name of server process
INT     PROCESS^NAMELEN;

BEGIN
   INT ERROR;
   INT ERROR^DETAIL;

!  Check whether process already running.  If so, open it.
!  If not, create it and open it:

   ERROR := PROCESS_GETPAIRINFO_(
                    !process^handle!,
                    PROCESS^NAME:PROCESS^NAMELEN);

!  If the process exists, open the server:

   CASE ERROR OF
   BEGIN

      0,  4 -> BEGIN

         !  The process already exists; open it:

         CALL OPEN^SERVER(PROCESS^NAME,PROCESS^NAMELEN,
                          SERVER^NUM)

      END;

      9 -> BEGIN

      !  The process does not exist; create it and open it,
      !  send it a Startup message, close it, and then reopen
      !  it:
```

```
!  Create process:

   ERROR := PROCESS_CREATE_(
                   SERVER^OBJECT^NAME:OBJFILE^NAMELEN,
                   !library^filename:library^file^len!,
                   !swap^filename:swap^file^len!,
                   !ext^swap^file^name:ext^swap^len!,
                   !priority!,
                   !processor!,
                   !process^handle!,
                   ERROR^DETAIL,
                   ZSYS^VAL^PCREATOPT^NAMEINCALL,
                   PROCESS^NAME:PROCESS^NAMELEN);
   IF ERROR <> 0 THEN
   BEGIN
      PRINT^STR("Unable to create server process");
      CALL PROCESS_STOP_;
   END;

!  Open the new server process:

   CALL OPEN^SERVER(PROCESS^NAME,PROCESS^NAMELEN,
                    SERVER^NUM);

!  Send the server a Startup message:

   START^UP^MESSAGE.MSG^CODE := -1;
   CALL WRITEX(SERVER^NUM,START^UP^MESSAGE,
               MESSAGE^LEN);
      IF <> THEN
      BEGIN
         CALL FILE_GETINFO_(SERVER^NUM,ERROR);
         IF ERROR <> 70 THEN
         BEGIN
            START^LINE;
            PUT^STR("Could not write Startup message");
            PUT^STR(" to server");
            PRINT^LINE;
            CALL PROCESS_STOP_;
         END;
      END;

!  Close the server:

   ERROR := FILE_CLOSE_(SERVER^NUM);

!  Reopen the server:

   CALL OPEN^SERVER(PROCESS^NAME,PROCESS^NAMELEN,
                    SERVER^NUM);
END;
```

```
      OTHERWISE -> BEGIN

      !  Unexpected error return from PROCESS_GETPAIRINFO_:

         PRINT^STR("Unexpected error ");
      END;

   END;
END;


!------------------------------------------------------------
! Procedure to process an invalid command.  The procedure
! informs the user that the selection was other than "r,"
! "p," "q," or "x."
!------------------------------------------------------------

PROC INVALID^COMMAND;
BEGIN

   PRINT^BLANK;

!  Inform the user that the selection was invalid and then
!  return to prompt again for a valid function:

   PRINT^STR ("INVALID COMMAND: " &
              "Type either 'r,' 'p,' 'q,' or 'x'");
END;


!------------------------------------------------------------
! Procedure to save the Startup message
!------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,
              LENGTH,MATCH) VARIABLE;

INT .RUCB,
    .START^DATA,
    .MESSAGE,
     LENGTH,
     MATCH;

BEGIN

!  Copy the Startup message into the START^UP^MESSAGE
!  structure and save the message length:

   START^UP^MESSAGE.MSG^CODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;
```

```
!------------------------------------------------------------
! Procedure to initialize the program. It calls
! INITIALIZER to save the Startup message.  It opens the
! terminal using the IN file from the Startup message and
! calls CREATE^AND^OPEN^SERVER to create and open each of
! the server processes.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING .OBJECT^FILE[0:MAXFLEN - 1]; !server object file
                                       ! name
   INT     OBJFILELEN;
   STRING .SERVER^NAME[0:MAXFLEN - 1]; !process name for
                                       ! servers
   INT     SERVERLEN;
   STRING .TERM^NAME[0:MAXFLEN - 1];   !file name for
                                       ! terminal
   INT     TERMLEN;
   INT     ERROR;


!  Read and process Startup message:

   CALL INITIALIZER(!rucb!,
                    !specifier!,
                    START^IT);

!  Open the home terminal.  Convert the IN file from the
!  Startup message into a D-series file name, and then open
!  it:

   !try with this temporary code

   ERROR := OLDFILENAME_TO_FILENAME_(START^UP^MESSAGE.INFILE,
                                     TERM^NAME:MAXFLEN,
                                     TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open $SER1, create it if it does not already exist:

   SERVER^NAME ':=' "$SER1" -> @S^PTR;
   SERVERLEN := @S^PTR '-' @SERVER^NAME;
   OBJECT^FILE ':=' "=SER1" -> @S^PTR;
   OBJFILELEN := @S^PTR '-' @OBJECT^FILE;
   CALL CREATE^AND^OPEN^SERVER(SERV1^NUM,OBJECT^FILE,
                    OBJFILELEN,SERVER^NAME,SERVERLEN);
```

```
!  Open $SER2, create it if it does not already exist:

   SERVER^NAME ':=' "$SER2" -> @S^PTR;
   SERVERLEN := @S^PTR '-' @SERVER^NAME;
   OBJECT^FILE ':=' "=SER2" -> @S^PTR;
   OBJFILELEN := @S^PTR '-' @OBJECT^FILE;
   CALL CREATE^AND^OPEN^SERVER(SERV2^NUM,OBJECT^FILE,
                    OBJFILELEN,SERVER^NAME,SERVERLEN);

!  Open $SER3, create it if it does not already exist:

   SERVER^NAME ':=' "$SER3" -> @S^PTR;
   SERVERLEN := @S^PTR '-' @SERVER^NAME;
   OBJECT^FILE ':=' "=SER3" -> @S^PTR;
   OBJFILELEN := @S^PTR '-' @OBJECT^FILE;
   CALL CREATE^AND^OPEN^SERVER(SERV3^NUM,OBJECT^FILE,
                    OBJFILELEN,SERVER^NAME,SERVERLEN);
END;


!-------------------------------------------------------------
! This is the main procedure.  It calls the INIT procedure to
! initialize and then goes into a loop calling GET^COMMAND
! to get the next user request and then calls a procedure
! to carry out the selected request.
!-------------------------------------------------------------

PROC REQUESTER MAIN;
BEGIN
   STRING CMD;

!  Perform initialization:

   CALL INIT;

!  Loop indefinitely until user selects function "x":

   WHILE 1 DO
   BEGIN

   !  Prompt for the next command:

      CMD := GET^COMMAND;
      !  Call the function selected by user:

         CASE CMD OF
         BEGIN

            "r", "R"  -> CALL READ^PART;

            "p", "P"  -> CALL PROCESS^ORDER;

            "q", "Q"  -> CALL READ^ORDER;

            "x", "X"  -> CALL EXIT^PROGRAM;

            OTHERWISE -> CALL INVALID^COMMAND;
         END;
      END;
   END;
```

22 Writing a Server Program

This section describes programming techniques that are useful when writing server programs. With Section 21, Writing a Requester Program, this section summarizes many of the techniques and procedures described earlier in this manual. You should be familiar with the information contained in Sections 1 through 20 before reading this section.

In addition to introducing the functions of a server process and some of the more common programming models, this section also describes how to add security to your requester/server application by limiting the number of requesters that can open a server and keeping track of which requesters have the server open.

The last part of this section provides sample server programs that form part of an application with the requester program described and shown in Section 21, Writing a Requester Program.

Functions of a Server Process

The most common use for server processes is in database applications, where a server process provides a database service. This service is usually to perform some application-dependent function on the database, such as reading a record or updating an account.

Sections 2 through 5 of this guide describe how to access data files using Guardian and Enscribe procedure calls. How a server reads requests from the requester using the $RECEIVE file is described in Section 6, Communicating With Processes.

Note. Not all servers are database servers. The requester/server application design model is general and can be used wherever it is desirable to separate functions of the application into different processes.

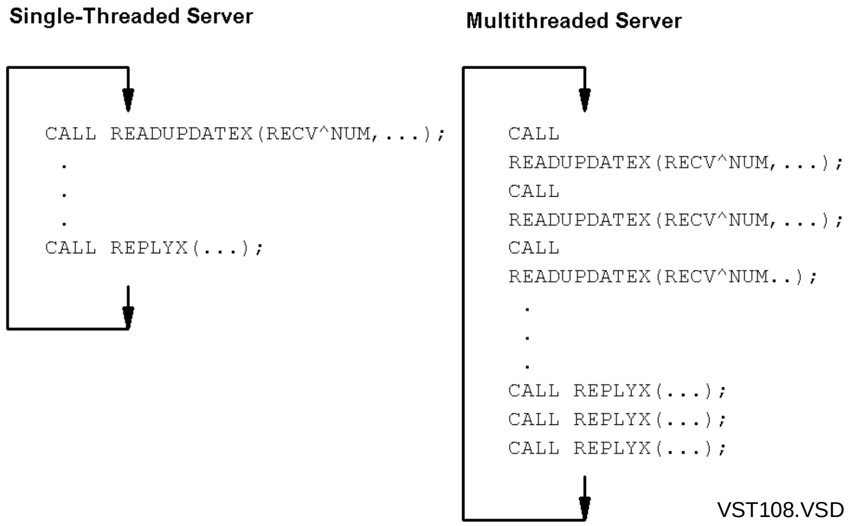
Multithreaded and Single-Threaded Servers

Servers can be single-threaded or multithreaded. In the single-threaded case, the server process reads a message from $RECEIVE, processes it, replies to it, and then reads the next message.

In a multithreaded server, the server process can read several messages and process them concurrently. To reply to the correct message, the server uses message tags.

Figure 22-1 shows the difference between single-threaded and multithreaded servers. For details of multithreaded servers, see Section 6, Communicating With Processes.

Figure 22-1. Single-Threaded and Multithreaded Servers



Single-Threaded Server

```
CALL READUPDATEX(RECV^NUM,...);
 .
 .
 .
CALL REPLYX(...);
```

Multithreaded Server

```
CALL
READUPDATEX(RECV^NUM,...);
CALL
READUPDATEX(RECV^NUM,...);
CALL
READUPDATEX(RECV^NUM..);
  .
  .
  .
CALL REPLYX(...);
CALL REPLYX(...);
CALL REPLYX(...);
```

VST108.VSD

Receive-Depth

The *receive-depth* parameter is used by servers for opening $RECEIVE. This parameter determines how many times the server can call READUPDATE before it has to call REPLY.

Receive-depth value	Description
0	Indicates that no calls to READUPDATE can be made. The server must use READ to receive messages. REPLY is not allowed (the file system completes the message exchange as soon as the message is read).
1	Is used for single-threaded servers (only one message is processed at a time by the server). In this case, if the requester has an active TMF transaction at the time the message is sent, the transaction is automatically made current in the server. When REPLY is called, the transaction is reset.
> 1 (greater than 1)	Is used for multithreaded servers (multiple messages are processed simultaneously by the server). If a multithreaded server is used by requesters with active TMF transactions, the server must call ACTIVATERECEIVETRANSID and pass the message tag value returned by FILE_GETRECEIVEINFO_, in order to make a previous TMF transaction current. Note that even though the server can have up to *receive-depth* messages queued internally, it can have only one current TMF transaction at a time.

Note that COBOL85 programs do not support multithreaded servers. You cannot open $RECEIVE with *receive-depth* value greater than 1 unless you use the ENTER verb to call the file system FILE_OPEN_ procedure directly. (This is not recommended, because it might adversely interfere with the COBOL85 RTL I/O mechanisms.) See the *COBOL85 Manual* for details.

If you use a multithreaded server as a pathway serverclass, make sure that the value of TCP SERVER LINKDEPTH is less than or equal to the *receive-depth* value. Note that the LINKDEPTH value must be 1 in all other cases. For single-threaded servers, a LINKDEPTH value greater than 1 disturbs the automatic load balancing feature in Pathway.

Context-Free Servers

Generally, you should design your server processes to be context free. Such servers have special advantages in application design. If the server retains no context, then each requester can request service from a given server without concern for what the server has previously done. Moreover, if multiple servers with identical function are available, then a requester can ask for service from any such server.

Maintaining an Opener Table

You can provide security to a server process by using an opener table. This table is maintained by the server and contains a list of all processes that have the server open. This table provides two functions:

- It allows you to know the number of processes that have the server open.
- It allows you to check that each message received originated from a process that has the server open

The Opener Table

An opener table typically consists of a sequence of 22-word entries. Each 22-word entry either is null or contains, in the first 10 words, the process handle of a requester that has this server open. If the process handle of a requester is present, the entry also contains the file number that the requester is using for the open. (Using a file number allows a requester to open a server more than once.) Associated with the opener table is an integer variable indicating the current number of openers (entries in the table).

The following declaration describes a typical opener table. Here, the maximum length of the table is set by the literal MAX^OPENERS:

```
INT     NUMBER^OF^OPENERS;         !number of requesters that
                                   ! have the server open

!Opener table contains information about who has the server
!open:
STRUCT .OPENER^TABLE[1:MAX^OPENERS];
```

```
BEGIN
   INT PROCESS^HANDLE [0:9];  !process handle of opener
   INT RESERVED^HANDLE [0:9]; !reserved, filled with -1; this
                              ! field is required for
OPENER_LOST_
   INT FILE^NUMBER;           !file number used by opener
   INT RESERVED^FILE^NUMBER;  !for use with NonStop pairs
END;
```

An opener table entry can have other fields defined in addition to those shown above. For example, to support full NonStop operation with opens from process pairs, the opener table entry would use fields, where the reserved areas are shown, for the backup open from a process pair; it would also have a field for the **sync ID** value. However, if neither NonStop operation nor opener context is to be supported, it is simpler to use only the two fields defined above, which can support backup opens by treating them as independent opens. The examples in this section use this simpler approach.

Getting Message Information

Before referring to your opener table, you first need to analyze the message read from $RECEIVE to determine what to do:

- If the message read from $RECEIVE is an Open message, you need to add the process handle of the requester to the opener table.
- If the message read from $RECEIVE is a Close message, you need to remove the process handle of the requester from the opener table.
- If the message read from $RECEIVE is a user request message, you need to check that the process handle of the requester is in the opener table before processing the request.

You can determine whether the received message is a system message or a user message by calling the FILE_GETINFO_ procedure immediately after reading from $RECEIVE. If the error number returned by FILE_GETINFO_ is 6, then the message is a system message. If the error number is 0, then the message is a user message.

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT);
CALL FILE_GETINFO_(RECV^NUM,ERROR);
IF ERROR = 6 THEN...           !system message
IF ERROR = 0 THEN...           !user message
```

In addition to checking for a system message, you should also get the process handle and file number of the process that sent the message. You will use the process handle and file number for adding an entry to the opener table or for comparing with entries that already exist in the opener table.

You can determine the process handle and file number of the message sender by calling the FILE_GETRECEIVEINFO_ procedure. This procedure returns 17 words of information about the message. Words 6 through 15 of the returned information contain the process handle of the message sender; word 3 contains the file number:

```
CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);
PROCESS^HANDLE ':=' RECEIVE^INFO[6] for 10;
FILE^NUMBER := RECEIVE^INFO[3];
```

Adding a Requester to the Opener Table

If the message received on $RECEIVE is an Open message (system message -103), then your program must try to add the requester to the opener table. First, you must scan the table, looking for a blank entry; then you can write the process handle into that space.

If the table is full, then you should reject the open attempt by returning error 12 to the requester process.

The following code attempts to add a process handle to the opener table. It assumes that a blank entry in the opener table contains -1 in each word.

```
!For an Open message (using literal from ZSYSTAL file):
IF BUFFER[0] = ZSYS^VAL^SMSG^OPEN THEN
BEGIN

   !Return "file in use" error if opener table is full:
   ERROR^NUMBER := 12;

   !Put the process ID into the opener table at the first
   !empty location and increment the count of openers.  Note
   !that you need check only the first word of the process
   !handle entry in the table; if it is -1, then the entry is
   !empty:
   I := 1;
   DONE := 0;
   WHILE I <= MAX^OPENERS AND DONE = 0 DO
   BEGIN
      IF OPENER^TABLE[I].PROCESS^HANDLE[0] = -1 THEN
      BEGIN
         OPENER^TABLE[I].PROCESS^HANDLE[0] ':='
           CALLING^PROCESS^PID FOR 10 WORDS;
         OPENER^TABLE[I].FILE^NUMBER := CALLING^PROCESS^FNUM;
         NUMBER^OF^OPENERS :=
            NUMBER^OF^OPENERS + 1;
         ERROR^NUMBER := 0;
         DONE := -1;
      END;
      I := I + 1;
   END;

   WCOUNT := 0;
   CALL REPLY(BUFFER,
              WCOUNT,
              !count^written!,
              !message^tag!,
              ERROR^NUMBER);
END;
```

Checking a Request Against the Opener Table

If the message received on $RECEIVE is a user message, then you need to check that the sender of the message has the server open. To do this, you scan each entry of the opener table looking for a match with the process handle and file number of the message sender. If no match is found, then you should reject the user request with error number 60. If a match is found, process the message.

The following example checks the opener table for a match:

```
I := 1;
WHILE I <= MAX^OPENERS DO
BEGIN
   IF RECEIVE^INFO[6] '='
         OPENER^TABLE[I].PROCESS^HANDLE[0] FOR 10 WORDS
         AND RECEIVE^INFO[3] = OPENER^TABLE[I].FILE^NUMBER THEN
   BEGIN

      !Process the user message
        .
        .
        .
       ERROR^NUMBER := 0;
       RETURN;
   END;
   I := I + 1;
END;
ERROR^NUMBER := 60;
```

Deleting a Requester From the Opener Table

The server process must delete the requester from its opener table whenever a requester no longer needs the service. The following situations can cause this:

- The requester closes the server.
- The CPU on which a requester is running fails.
- The network connection between the requester and server fails.

When the Requester Closes the Server

When a requester process closes a server, the server receives a Close message (system message -104). On receipt of this message, the server must delete the corresponding entry from the opener table. It does this by finding the process handle and file number in the opener table and then deleting the entry by writing -1 over each word.

When a CPU or Network Connection Fails

To be able to delete an entry from the opener table when the CPU of an opener fails or when the network connection between the requester and server fails, your server process must do the following:

- Call the MONITORCPUS procedure so that the server will receive CPU down messages (system message -2) on $RECEIVE
- Call the MONITORNET procedure so that the server will receive Remote CPU down (system message -100) and Node Down (system message -110) messages
- Check $RECEIVE periodically for receipt of the CPU down, Remote CPU down, or Node down message
- On receipt of a CPU down, Remote CPU down, or Node down message, scan the opener table for openers that were running on the failed CPU
- Delete the opener from the opener table

The following example performs these tasks. It uses the OPENER_LOST_ procedure to check system messages for information about lost openers. It deletes openers from the opener table if the received message is a processor down (system message -2), Remote processor down (system message -100), or Node Down (system message -110) message.:

```
!Check for processor down messages from all local processor
!modules:
CPU^MASK.:= -1;
CALL MONITORCPUS(CPU^MASK);

!Check for failure of a remote processor or a remote node, or
!for failure to communicate with the remote node:
CALL MONITORNET(1);
 .
 .

!Read from $RECEIVE:
CALL READX(RECV^NUM,BUFFER,BUFSIZE,COUNT^READ);
IF <> THEN
BEGIN
   CALL FILE_GETINFO_(RECV^NUM,ERROR);
   IF ERROR = 6 THEN             !system message received
   BEGIN
      .
      !Check for lost openers:
      INDEX := -1;

      DO
      BEGIN
         STATUS := OPENER_LOST_(BUFFER:COUNT^READ,
                                OPENER^TABLE[1],
                                INDEX,
                                MAX^OPENERS,
                                $LEN(OPENER^TABLE[1]));
         IF STATUS = 6 THEN
            NUMBER^OF^OPENERS := NUMBER^OF^OPENERS - 1;
      END
      UNTIL STATUS = 0 OR STATUS = 2
            OR STATUS = 3 OR STATUS = 7;
      .
      .
      !Process other system messages.
      .
      .
   END;
```

Writing a Server Program: An Example

The sample server programs given in this section provide service to the requester program described in Section 21, Writing a Requester Program. The servers and the requester together provide the following application functions:

- Queries the database to find out how much of a given item is on hand.
- Processes an order by updating the inventory database and creating an order record.
- Queries the status of an existing order to find out who placed the order, when the order was placed, and whether the order has been shipped.

A separate server process provides database service for each of the above functions.

Application Overview

The application database is made up of part records and order records. The part records are contained in the inventory file and the order records in the orders file.

The Inventory File

The inventory file contains one record for each item that the store carries. A part record contains the following information about a given part:

- The part number
- A brief description of the item
- The quantity of the item currently on hand

- The unit price of the item
- The name of the supplier
- If an order has been placed with the supplier, the quantity ordered and the expected delivery date

The Orders File

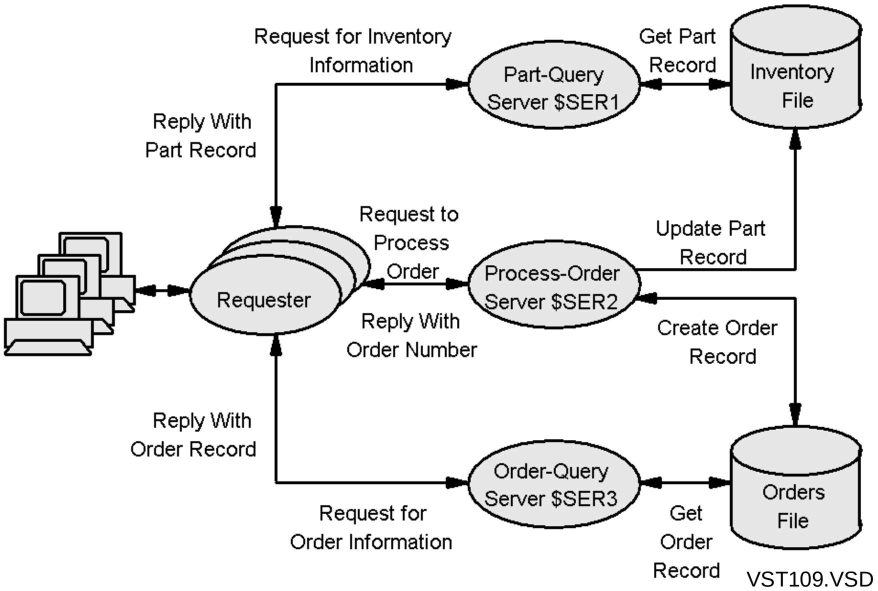
The orders file contains one order record for each item ordered by a customer. The orders file contains the following information:

- The part number of the item ordered
- A brief description of the item ordered
- The quantity ordered
- The name, address, and credit-card number of the customer
- The date when the order was placed
- The date that the order was shipped (if it has been shipped)
- The status of the order, indicating whether the order has been shipped, paid for, and so on

The Role of the Server Processes in the Application

Figure 22-2 shows the role of server processes in the application.

Figure 22-2. Server Processes in the Example Application


Request for Inventory Information
Get Part Record
Part-Query Server $SER1
Inventory File
Reply With Part Record
Request to Process Order
Update Part Record
Requester
Process-Order Server $SER2
Reply With Order Number
Create Order Record
Reply With Order Record
Order-Query Server $SER3
Orders File
Request for Order Information
Get Order Record
VST109.VSD

The requester chooses the server to send a request to depending on the function requested by the user:

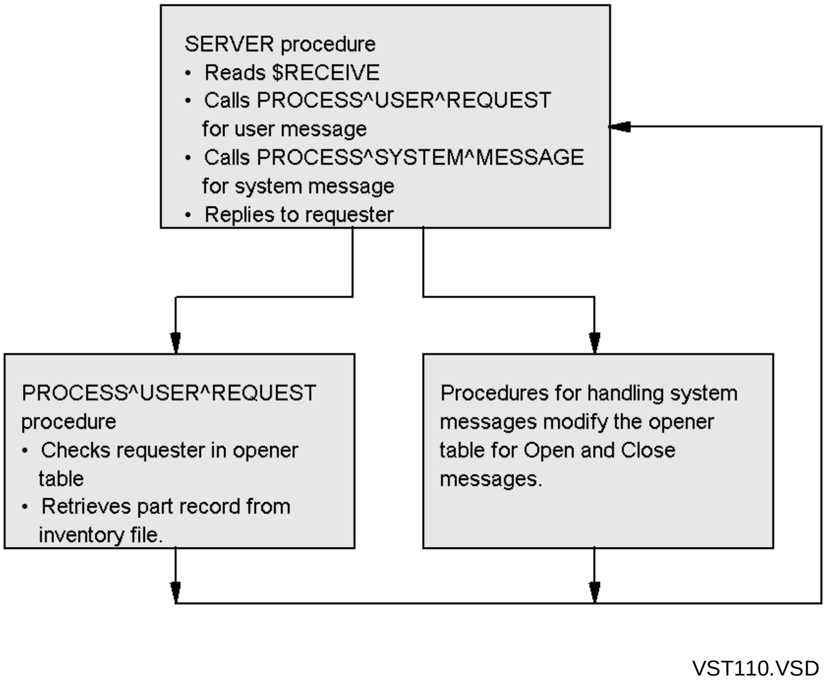
- If the user requests to query a part record, then the requester obtains the part record by sending a request to the part-query server ($SER1). The $SER1 process obtains the information from the inventory file and returns it to the requester.
- If the user requests to process an order, the requester sends a message to the process-order server ($SER2). This server uses the information it receives to update the inventory level in the inventory file and to create an order record and put it in the orders file. The server returns the order number to the requester.
- If the user wants to query an existing order record, then the requester sends a message to the order-query server ($SER3), which queries the orders file and then sends the corresponding order record back to the requester.

See Section 21, Writing a Requester Program, for a detailed description of the requester process.

The Part-Query Server ($SER1)

Figure 22-3 shows the internal function of the part-query server, in terms of its major procedures.

Figure 22-3. Relationship Between Major Procedures in the Part-Query Server


SERVER procedure
• Reads $RECEIVE
• Calls PROCESS^USER^REQUEST for user message
• Calls PROCESS^SYSTEM^MESSAGE for system message
• Replies to requester
PROCESS^USER^REQUEST procedure
• Checks requester in opener table
• Retrieves part record from inventory file.
Procedures for handling system messages modify the opener table for Open and Close messages.
VST110.VSD

The following paragraphs describe the major procedures in detail.

The SERVER Procedure

The SERVER procedure is the main procedure for the part-query server. It provides three main functions:

- Calls INIT to perform server initialization
- Calls a procedure based on messages received on $RECEIVE
- Replies to the message read from $RECEIVE

The server-initialization phase performed by INIT involves reading the Startup message, opening the home terminal, and opening the inventory file.

The rest of the SERVER procedure responds to messages read from $RECEIVE. If the message is a system message, then the procedure calls the PROCESS^SYSTEM^MESSAGE procedure. If the message received is a user message, then the procedure calls the PROCESS^USER^REQUEST procedure. Before calling either of these procedures, the SERVER procedure calls the FILE_GETRECEIVEINFO_ system procedure to get the process handle of the process that sent the message. This process handle is used later for controlling access to the server.

On return from either the PROCESS^SYSTEM^MESSAGE or PROCESS^USER^REQUEST procedure, the SERVER procedure replies to the message. For a user message, the reply consists of a part record or an error indication. For a system message, the reply consists of an error indication: 0 for a successful operation or some positive number for an unsuccessful operation.

Procedures for Handling System Messages

Procedures for handling system messages include the PROCESS^SYSTEM^MESSAGE, PROCESS^OPEN^MESSAGE, PROCESS^CLOSE^MESSAGE, and PROCESS^OTHER^MESSAGE procedures. The PROCESS^SYSTEM^MESSAGE procedure is called from the SERVER procedure whenever the server reads a system message from the $RECEIVE file. PROCESS^SYSTEM^MESSAGE calls one of the other procedures for handling system messages, depending on whether the system message is an Open message, a Close message, or some other system message.

- If the system message is an Open message, then PROCESS^SYSTEM^MESSAGE calls the PROCESS^OPEN^MESSAGE procedure. This procedure tries to add an entry to the server’s opener table. It checks each 10-word entry in turn until it finds a blank entry (consisting of a -1 in each word). The procedure then copies the process handle of the requester into the blank entry and returns to the SERVER procedure with error number zero.

  If the opener table is full, then the process returns error 12 to the SERVER procedure.

- If the system message is a Close message, then PROCESS^SYSTEM^MESSAGE calls the PROCESS^CLOSE^MESSAGE procedure to check that the process handle of the sending process exists in the opener table and to remove the entry. If the process handle exists in the opener table, then error number zero is returned to the SERVER process; otherwise, error 60 is returned.

- If the system message is any system message other than Open or Close, then PROCESS^SYSTEM^MESSAGE calls the PROCESS^OTHER^MESSAGE procedure to make updates to the opener table if the message concerns a network connection or CPU failure.

The PROCESS^USER^REQUEST Procedure

The PROCESS^USER^REQUEST procedure is called by the SERVER procedure when a user message is read from the $RECEIVE file. Its function is to read a specified record from the inventory file.

First, the PROCESS^USER^REQUEST procedure checks each entry in the opener table to see whether the sender of the user message has this server open. If not, then the procedure returns error number 60 to the SERVER procedure.

If the requesting procedure is in the server’s opener table, then the PROCESS^USER^REQUEST procedure uses the part number provided in the user message as a key to the inventory file to access the desired record. If the record exists, then the record is returned to the SERVER procedure with an error condition of zero. If the record does not exist, then the file-system error number is returned without a part record.

The Code for the Part-Query Server ($SER1)

The code for the part-query server program appears on the following pages.

```
?INSPECT, SYMBOLS, NOCODE
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!-----------------------
!Literals:
!-----------------------

LITERAL MAX^OPENERS = 2,        !maximum number of openers
                                ! allowed
        EXACT       = 2,        !for exact key positioning
        MAXFLEN     = ZSYS^VAL^LEN^FILENAME,
                                !maximum length for file name
        BUFSIZE     = 512;      !size of I/O buffer

!------------------------
!Global data structures:
!------------------------

!Data structure for Startup message:

STRUCT .START^UP^MESSAGE;
BEGIN
   INT     MSG^CODE;
   STRUCT  DEFAULT;
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
   END;
   STRUCT  INFILE;              !IN file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
   STRUCT  OUTFILE;             !OUT file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
   STRING  PARAM[0:529];        !parameter string
END;
INT    MESSAGE^LEN;             !length of Startup message
```

```
!Message received from requester.  Contains a part number:

STRUCT PART^REQUEST;
BEGIN
   STRING PART^NUMBER[0:9];     !10-digit part number
END;


!Message returned to requester.  Contains part record
!information obtained from the inventory file:

STRUCT .PART^REC;
BEGIN
   STRING PART^NUMBER[0:9];     !10-digit part number
   STRING PART^DESC[0:47];      !description of part
   STRING SUPPLIER[0:23];       !name of part supplier
   INT QUANTITY^ON^HAND;        !how many of this part on
                                ! hand
   INT UNIT^PRICE;              !cost of one part in dollars
   INT ORDER^PLACED[0:2];       !date when part last ordered
                                ! from supplier
   INT SHIPMENT^DUE[0:2];       !date shipment due from
   INT QUANTITY^ORDERED;        !how many ordered from
END;                            ! supplier


!Data structure for the opener table:

STRUCT .OPENER^TABLE;           !information about who has
BEGIN                           ! the server open
   INT CURRENT^COUNT;           !how many requesters have
                                ! this server open
   STRUCT OCB[1:MAX^OPENERS];   !one entry for each opener
   BEGIN

      !Process handle of an opener:

      INT PROCESS^HANDLE[0:9];    !process handle of opener
      INT RESERVED^HANDLE[0:9];   !reserved, filled with -1
      INT FILE^NUMBER;            !file number used by opener
   END;
END;
```

```
!------------------------
!Other global variables:
!------------------------

STRING .S^PTR;                      !pointer to end of string

INT     TERM^NUM;                   !file number for terminal
INT    .BUFFER[0:BUFSIZE/2 - 1];!I/O buffer
STRING .SBUFFER := @BUFFER[0] '<<' 1; !string pointer to I/O
                                    ! buffer
INT     REPLY^ERROR;                !error value returned to
                                    ! requester
INT     INV^FNUM;                   !file number for inventory
                                    ! file
INT     REPLY^LEN;                  !length of reply buffer
INT     RECV^NUM;                   !file number for $RECEIVE
INT    .RECEIVE^INFO[0:16];         !returned by
                                    ! FILE_GETRECEIVEINFO_

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,
?               FILE_GETINFO_,PROCESS_STOP_,
?               FILE_GETRECEIVEINFO_,KEYPOSITION,DNUMOUT,
?               WRITEX,OLDFILENAME_TO_FILENAME_,READUPDATEX,
?               REPLYX)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =    @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N)  =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE  PRINT^LINE =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

  DEFINE  PRINT^STR(S) =  BEGIN  START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;   END;  #;
```

```
!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name and its length
! and the error number.  This procedure is used when the
! file is not open, so there is no file number for it.
!
! The procedure also stops the program after displaying the
! error message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error from $SER1 ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number,
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT   FNUM;
BEGIN
   INT     ERROR;
   STRING .FNAME[0:MAXFLEN-1];
   INT     FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!---------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!---------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);
END;
```

```
!------------------------------------------------------------
! Procedure to process a request for a part record.  This
! procedure checks that the process that sent the message is
! in the opener table before retrieving the part record from
! the inventory file using the key supplied in the part
! number.
!------------------------------------------------------------

PROC PROCESS^USER^REQUEST;

BEGIN
   INT   POSITIONING^MODE;   !used by KEYPOSITION
   INT   COUNT^READ;
   INT   COUNT;
   INT   J;
   INT   I;

!  Check that the process handle of the requester is in the
!  opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN AND
          RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
      BEGIN

      !  Copy user message from requester into data
      !  structure:

         PART^REQUEST.PART^NUMBER ':=' SBUFFER[0] FOR 10;

      !  Position pointers to appropriate record, based on
      !  the key value supplied in the request:

         POSITIONING^MODE := EXACT;
         CALL KEYPOSITION(INV^FNUM,
                          PART^REQUEST.PART^NUMBER,
                          !key^specifier!,
                          !length^word!,
                          POSITIONING^MODE);
         IF <> THEN CALL FILE^ERRORS(INV^FNUM);

      !  Read the record from the inventory file:

         CALL READUPDATEX(INV^FNUM,SBUFFER,BUFSIZE,COUNT^READ);
      !  If unable to position to the requested key, return
      !  the error number.  This error occurs when the key is
      !  not in the inventory file.

         IF <> THEN
         BEGIN
            CALL FILE_GETINFO_(INV^FNUM,REPLY^ERROR);
            RETURN;
         END;

      !  Clear the REPLY^ERROR variable if the read is
      !  successful:

         REPLY^LEN := $LEN(PART^REC);
         REPLY^ERROR := 0;
         RETURN;
      END;

   !  Check next entry in the opener table:
      I := I + 1;
   END;

!  Requester not in opener table:
   REPLY^ERROR := 60;
END;


!------------------------------------------------------------
! Procedure to process an Open system message (-103).  It
! places the process handle of the requester in the opener
! table, if there is room.  If the table is full, it
! rejects the open.
!------------------------------------------------------------

PROC PROCESS^OPEN^MESSAGE;

BEGIN
   INT I;
   INT J;
   INT COUNT;

!  Check if opener table full.  Return "file in use" error if
!  it is full:

   IF OPENER^TABLE.CURRENT^COUNT >= MAX^OPENERS THEN

   BEGIN
      REPLY^ERROR := 12;
      RETURN;
   END;
```

```
!  Put the process handle into the opener table at the first
!  empty location and increment the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF OPENER^TABLE.OCB[I].PROCESS^HANDLE[J] = -1
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN THEN
      BEGIN
         OPENER^TABLE.OCB[I] ':='
                RECEIVE^INFO[6] FOR ZSYS^VAL^PHANDLE^WLEN;
         OPENER^TABLE.OCB[I].FILE^NUMBER := RECEIVE^INFO[3];
         OPENER^TABLE.CURRENT^COUNT :=
                OPENER^TABLE.CURRENT^COUNT + 1;
         REPLY^LEN := 0;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure to process a Close system message.  It removes
! the requester from the opener table.
!------------------------------------------------------------

PROC PROCESS^CLOSE^MESSAGE;

BEGIN
   INT   I;
   INT   J;
   INT   COUNT;

!  Check that the closing process is in the opener table.
!  If so, remove the entry from the opener table and
!  decrement the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN AND
         RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
      BEGIN
         OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                          ZSYS^VAL^PHANDLE^WLEN * [-1];
         OPENER^TABLE.CURRENT^COUNT :=
                          OPENER^TABLE.CURRENT^COUNT - 1;
         REPLY^LEN := 0;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;

!  If calling process not in opener table, return error 60:

   REPLY^ERROR := 60;
END;
```

```
!------------------------------------------------------------
! Procedure to process a system message other than Open or
! Close.
!------------------------------------------------------------

PROC PROCESS^OTHER^MESSAGE;

BEGIN

INT INDEX, STATUS;
INDEX := -1;
   DO BEGIN
      STATUS := OPENER_LOST_( BUFFER:COUNT^READ,
                OPENER^TABLE.OCB[1], INDEX,
                MAX^OPENERS, $LEN( OPENER^TABLE.OCB[1] ));
      IF STATUS = 6 THEN
         OPENER^TABLE.CURRENT^COUNT :=
                        OPENER^TABLE.CURRENT^COUNT - 1;
      END
   UNTIL STATUS = 0 OR STATUS = 2 OR STATUS 3 OR STATUS = 7;
   REPLY^ERROR := 0;
   REPLY^LEN := 0;

END;


!------------------------------------------------------------
! Procedure to process a system message.
!------------------------------------------------------------

PROC PROCESS^SYSTEM^MESSAGE;

BEGIN
   CASE BUFFER[0] OF
   BEGIN

      -103      -> CALL PROCESS^OPEN^MESSAGE;

      -104      -> CALL PROCESS^CLOSE^MESSAGE;

      OTHERWISE -> CALL PROCESS^OTHER^MESSAGE;
   END;
END;
```

```
!--------------------------------------------------------------
! Procedure to save the Startup message.
!--------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,
              MATCH) VARIABLE;
INT      .RUCB,
         .START^DATA,
         .MESSAGE,
          LENGTH,
          MATCH;

BEGIN

!  Copy the Startup message into the START^UP^MESSAGE
!  structure and save the message length:

   START^UP^MESSAGE.MSG^CODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;


!--------------------------------------------------------------
! Procedure to perform initialization.  It calls INITIALIZER
! to read the Startup message then opens the IN file, the
! inventory file, and $RECEIVE, and then initializes the
! opener table.
!--------------------------------------------------------------

PROC INIT;
BEGIN
   STRING   .TERM^NAME[0:MAXFLEN - 1]; !terminal file name
   INT       TERMLEN;
   STRING   .RECV^NAME[0:MAXFLEN - 1]; !$RECEIVE file name
   STRING   .INV^FNAME[0:MAXFLEN - 1]; !data file name
   INT       INV^FLEN;
   INT       RECV^DEPTH;               !receive depth
   INT       I;
   INT       ERROR;

!  Read the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Open the home terminal (IN file);

   ERROR := OLDFILENAME_TO_FILENAME_(START^UP^MESSAGE.INFILE,
                                     TERM^NAME:MAXFLEN,
                                     TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
```

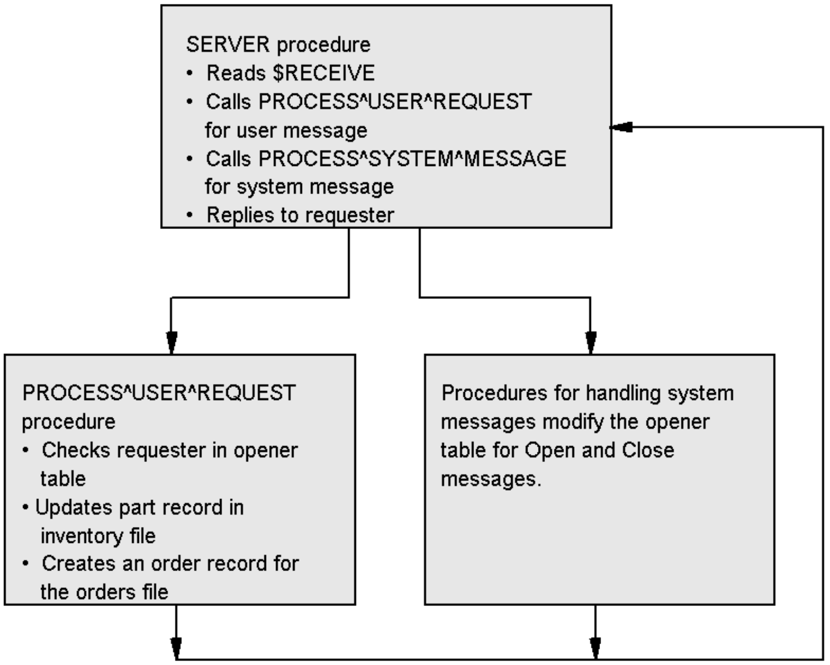
```
!  Open $RECEIVE with a receive depth of 1 and to accept
!  system messages (the default):

   RECV^NAME ':=' "$RECEIVE" -> @S^PTR;
   RECV^DEPTH := 1;
   ERROR := FILE_OPEN_(RECV^NAME:@S^PTR '-' @RECV^NAME,
                       RECV^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       RECV^DEPTH);

!  Instruct the operating system to send status change messages
!  for processors in both local and remote systems.

   CALL MONITORCPUS( -1 );
   CALL MONITORNET( 1 );

!  Open the INVENTORY file:

   INV^FNAME ':=' "=INV^FNAME" -> @S^PTR;
   INV^FLEN := @S^PTR '-' @INV^FNAME;
   ERROR := FILE_OPEN_(INV^FNAME:INV^FLEN,INV^FNUM);
   IF ERROR <> 0 THEN
         CALL FILE^ERRORS^NAME(INV^FNAME:INV^FLEN,ERROR);

!  Initialize the opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                             [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      OPENER^TABLE.OCB[I].RESERVED^HANDLE ':='
                             [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Main procedure calls INIT to perform initialization and
! then goes into a loop in which it reads the $RECEIVE file.
! It calls the appropriate procedure depending on whether the
! message read was a system message, a user message, or
! whether the read operation generated an error.
!------------------------------------------------------------

PROC SERVER MAIN;
BEGIN
   INT COUNT^READ;
   INT ERROR;

!  Initialize files and opener table:

   CALL INIT;

!  Loop forever:

   WHILE 1 DO
   BEGIN

   !  Read a message from $RECEIVE and check for an error:

      CALL READUPDATEX(RECV^NUM,SBUFFER,BUFSIZE,COUNT^READ);
      CALL FILE_GETINFO_(RECV^NUM,ERROR);

   !  Get the process handle of the requesting process:

      CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);

   !  Select a procedure depending on the results of the
   !  read operation:

      CASE ERROR OF
      BEGIN

      !  For a user message, call the PROCESS^USER^REQUEST
      !  procedure:

         0 -> CALL PROCESS^USER^REQUEST;

      !  For a system message, call the
      !  PROCESS^SYSTEM^MESSAGE procedure:

         6 -> CALL PROCESS^SYSTEM^MESSAGE;
      !  For any other error return, call the FILE^ERRORS
      !  procedure:

         OTHERWISE -> CALL FILE^ERRORS(RECV^NUM);
      END;

   !  Reply to the message:

      CALL REPLYX(SBUFFER,
                  REPLY^LEN,
                  !count^written!,
                  !message^tag!,
                  REPLY^ERROR);
   END;
END;
```

The Process-Order Server ($SER2)

Figure 22-4 shows the function of each procedure in the process-order server and the relationships among the procedures.

Figure 22-4. Relationship Between Major Procedures in the Process-Order Server


SERVER procedure
• Reads $RECEIVE
• Calls PROCESS^USER^REQUEST for user message
• Calls PROCESS^SYSTEM^MESSAGE for system message
• Replies to requester
PROCESS^USER^REQUEST procedure
• Checks requester in opener table
• Updates part record in inventory file
• Creates an order record for the orders file
Procedures for handling system messages modify the opener table for Open and Close messages.

VST111.VSD

As you can see from Figure 22-4, the structure of the process-order server is similar to that of the part-query server. The SERVER procedure and the procedures for handling system messages are the same as for the part-query server. The differences are in the PROCESS^USER^REQUEST procedure.

The user message read by the SERVER procedure contains information needed to process an order request. This information includes the part number and quantity of the requested item, and the name, address, and credit-card number of the customer.

If the requester process is in the opener table, then the PROCESS^USER^REQUEST procedure modifies the application database as follows:

- It retrieves the part record from the inventory file and checks that there are enough items in stock to satisfy the request. If so, the PROCESS^USER^REQUEST procedure updates the part record with the new stock level. If there is not enough stock to satisfy the request, then the procedure returns to the SERVER procedure with the new (negative) stock level; the inventory file does not get updated because the requester process uses the negative number to reject the order request.

- If there is enough stock on hand to satisfy the request, then the PROCESS^USER^REQUEST procedure creates an order record out of the information sent by the requester, adds the date of the order to the structure, and then writes the new record to the orders file.

The Code for the Process-Order Server ($SER2)

The code for the process-order server program appears on the following pages.

```
?INSPECT, SYMBOLS, NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

!-----------------------
!Literals:
!-----------------------

LITERAL MAX^OPENERS = 2,        !maximum number of openers
                                ! allowed
        EXACT       = 2,        !for exact key positioning
        MAXFLEN     = ZSYS^VAL^LEN^FILENAME,
                                !maximum length for file name
        BUFSIZE     = 512;

!------------------------
!Global data structures:
!------------------------

!Data structure for Startup message:

STRUCT .START^UP^MESSAGE;
BEGIN
   INT     MSG^CODE;
   STRUCT  DEFAULT;
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
   END;
   STRUCT  INFILE;              !IN file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
   STRUCT  OUTFILE;             !OUT file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
```

```
      STRING  PARAM[0:529];          !parameter string
   END;
   INT    MESSAGE^LEN;               !length of Startup message


   !Message received from requester.  Contains order
   !request information:

   STRUCT .ORDER^REQUEST;
   BEGIN
      STRUCT  NAME;                  !customer's name
      BEGIN
         STRING LAST[0:19];
         STRING FIRST[0:19];
         STRING INITIAL[0:1];
      END;
      STRING ADDRESS[0:47];          !customer's street address
      STRING CITY[0:23];             !city name
      STRING ZIP[0:7];               !customer's zip code
      STRING CCN[0:15];              !customer's credit-card
                                     ! number
      STRING PART^NUMBER[0:9];       !part number of item ordered
      STRING PART^DESC[0:47];        !description of item ordered
      INT    QTY^ORDERED;            !quantity of item ordered
   END;



   !Record to access orders file.  Contains information
   !about an order:

   STRUCT .ORDER^RECORD;
   BEGIN
      STRING ORDER^NUMBER[0:27];     !28-digit order number
      STRUCT  NAME;                  !customer's name
      BEGIN
         STRING LAST[0:19];
         STRING FIRST[0:19];
         STRING INITIAL[0:1];
      END;
      STRING ADDRESS[0:47];          !customer's street address
      STRING CITY[0:23];             !city name
      STRING ZIP[0:7];               !customer's zip code
      STRING CCN[0:15];              !customer's credit-card
                                     ! number
      STRING PART^NUMBER[0:9];       !part number of item ordered
      STRING PART^DESC[0:47];        !description of item ordered
      INT    QTY^ORDERED;            !quantity of item ordered
      INT    DATE^ORDERED[0:2];      !date that the order was
                                     ! placed
      INT    DATE^SHIPPED[0:2];      !date order shipped to
                                     ! customer
      STRING SHIPPING^STATUS[0:1];  !status of order
```

```
END;                              ! supplier


!Message returned to requester.  Contains the new stock
!level and the new order number:

STRUCT  .ORDER^REPLY;
BEGIN
   INT    QUANTITY^ON^HAND;
   STRING ORDER^NUMBER[0:27];
END;


!Record to access inventory file.  It contains information
!about a part record:

STRUCT  .PART^REC;
BEGIN
   STRING PART^NUMBER[0:9];      !10-digit part number
   STRING PART^DESC[0:47];       !description of part
   STRING SUPPLIER[0:23];        !name of part supplier
   INT    QUANTITY^ON^HAND;      !how many of this part on
                                 ! hand
   INT    UNIT^PRICE;            !cost of one part in dollars
   INT    ORDER^PLACED[0:2];     !date when part last ordered
                                 ! from supplier
   INT    SHIPMENT^DUE[0:2];     !date shipment due from
                                 ! supplier
   INT    QUANTITY^ORDERED;      !how many ordered from
END;                             ! supplier


!Data structure for the opener table:

STRUCT .OPENER^TABLE;            !information about who has
BEGIN                            ! the server open
   INT CURRENT^COUNT;            !how many requesters have
                                 ! this server open
   STRUCT OCB[1:MAX^OPENERS];    !one entry for each opener
   BEGIN

      !Process handle of an opener:

      INT PROCESS^HANDLE[0:9];    !process handle of opener
      INT RESERVED^HANDLE[0:9];   !reserved, filled with -1
      INT FILE^NUMBER;            !file number used by opener
   END;
END;
```

```
!------------------------
!Other global variables:
!------------------------

STRING .S^PTR;                  !pointer to end of string

INT     TERM^NUM;               !file number for terminal
INT    .BUFFER[0:BUFSIZE/2 - 1];!I/O buffer
STRING .SBUFFER := @BUFFER[0] '<<' 1; !string pointer to I/O
                                ! buffer
INT     REPLY^LEN;              !length of reply buffer
INT     REPLY^ERROR;            !error value returned to
                                ! requester
INT     ORD^FNUM;               !file number for orders file
INT     INV^FNUM;               !file number for inventory
                                ! file
INT     RECV^NUM;               !file number for $RECEIVE file
INT    .RECEIVE^INFO[0:16];     !returned by
                                ! FILE_GETRECEIVEINFO_

?NOLIST
?SOURCE
$SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,FILE_GETINFO_,
?                       PROCESS_STOP_,FILE_GETRECEIVEINFO_,
?                       KEYPOSITION,DNUMOUT,WRITEX,NUMOUT,
?                       OLDFILENAME_TO_FILENAME_,TIMESTAMP,
?                       REPLYX,READUPDATELOCKX,
?                       WRITEUPDATEUNLOCKX,UNLOCKREC,
?                       INTERPRETTIMESTAMP,JULIANTIMESTAMP,
?                       READUPDATEX)
?LIST
```

```
!------------------------------------------------------------
! Here are a few DEFINEs to make it a little easier to
! format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =    @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N)  =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE  PRINT^LINE =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

  DEFINE  PRINT^STR(S) =  BEGIN  START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;   END;  #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name and its length
! and the error number.  This procedure is used when the
! file is not open, so there is no file number for it.
!
! The procedure also stops the program after displaying the
! error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error from $SER1 ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT   FNUM;
BEGIN
   INT     ERROR;
   STRING .FNAME[0:MAXFLEN-1];
   INT     FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;
```

```
!-----------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!-----------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);
END;
```

```
!------------------------------------------------------------
! Procedure to process an order request.  This procedure
! checks that the process that sent the message is in the
! opener table before updating the part record with the new
! quantity on hand and creating an order record and writing
! it to the orders file.
!------------------------------------------------------------

PROC PROCESS^USER^REQUEST;

BEGIN
   INT     J^DATE^AND^TIME[0:7]; !for Gregorian date and time
   INT(32) JD^NUMBER;            !Julian day number
   FIXED   J^TIME;               !Julian timestamp
   INT     BASE, WIDTH;          !for NUMOUT procedure
   INT     POSITIONING^MODE;     !used by KEYPOSITION
   INT     COUNT^READ;
   INT     COUNT;
   INT     I, J, K, L;           !counters

!  Check that the process handle of the requester is in the
!  opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;

   !  If there is a match, set I to a value that will exit
   !  the loop:

   IF COUNT = ZSYS^VAL^PHANDLE^WLEN
      AND RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
   I := MAX^OPENERS;

   !  Check if last entry has just failed to match:

      IF ( COUNT <> ZSYS^VAL^PHANDLE^WLEN
         OR RECEIVE^INFO[3] <> OPENER^TABLE.OCB[I].FILE^NUMBER )
            AND I = MAX^OPENERS THEN
         !  Requester not in opener table:

            BEGIN
               REPLY^LEN := 0;
               REPLY^ERROR := 60;
               RETURN
            END;
            I := I + 1;
         END;

      !  Proceed because the requester is in the opener table.

      !-----------------------------
      !Update inventory record with
      !new quantity on hand and
      !prepare the reply structure
      !with new quantity information
      !-----------------------------

      !  Copy user message from requester into data
      !  structure:

         ORDER^REQUEST.NAME.LAST ':=' SBUFFER[0] FOR
                                      $LEN(ORDER^REQUEST);

      !  Position pointers to appropriate record, based on
      !  the key value supplied in the request:

         POSITIONING^MODE := EXACT;
         CALL KEYPOSITION(INV^FNUM,ORDER^REQUEST.PART^NUMBER,
                          !key^specifier!,
                          !length^word!,
                          POSITIONING^MODE);
         IF <> THEN CALL FILE^ERRORS(INV^FNUM);

      !  Read the record from the inventory file:

         CALL READUPDATELOCKX(INV^FNUM,SBUFFER,BUFSIZE,
                              COUNT^READ);

      !  If unable to position to the requested key, return
      !  the error number.  This error occurs when the key is
      !  not in the orders file.

         IF <> THEN
         BEGIN
            CALL FILE_GETINFO_(INV^FNUM,REPLY^ERROR);
            REPLY^LEN := 0;
            RETURN;
         END;

      !  Copy the contents of the buffer into the part record
      !  structure:

         PART^REC.PART^NUMBER ':=' SBUFFER FOR $LEN(PART^REC);
```

```
!  Check that there is still enough stock on hand to satisfy
!  the order:

   ORDER^REPLY.QUANTITY^ON^HAND := PART^REC.QUANTITY^ON^HAND
                              - ORDER^REQUEST.QTY^ORDERED;

!  If not enough stock, unlock the record and return with
!  negative stock indication:

   IF ORDER^REPLY.QUANTITY^ON^HAND < 0 THEN
   BEGIN
      CALL UNLOCKREC(INV^FNUM);
      SBUFFER[0] ':=' ORDER^REPLY FOR $LEN(ORDER^REPLY);
      REPLY^LEN := $LEN(ORDER^REPLY);
      REPLY^ERROR := 0;
      RETURN;
   END;

!  If there is enough stock, update the inventory record:

   PART^REC.QUANTITY^ON^HAND := ORDER^REPLY.QUANTITY^ON^HAND;
   CALL WRITEUPDATEUNLOCKX(INV^FNUM,PART^REC,$LEN(PART^REC));

!------------------------------
!Add new order record to orders
!file and prepare reply message
!with new order number.
!------------------------------

!  Blank the ORDER^RECORD structure:

   ORDER^RECORD.ORDER^NUMBER ':='
                      ($LEN(ORDER^RECORD) / 2) * ["  "];
   ORDER^RECORD.QTY^ORDERED := 0;
   K := 0;
   WHILE K < 3 DO
   BEGIN
      ORDER^RECORD.DATE^ORDERED[K] := 0;
      ORDER^RECORD.DATE^SHIPPED[K] := 0;
      K := K + 1;
   END;
```

```
!  Create an order number based on a Julian timestamp and
!  the part number.  The INTERPRETTIMESTAMP procedure
!  converts the timestamp into a Gregorian date and time,
!  which subsequent calls to NUMOUT convert into strings.
!  Note that in the year part, the first 2 digits are
!  truncated:

   J^TIME := JULIANTIMESTAMP;
   JD^NUMBER := INTERPRETTIMESTAMP(J^TIME,J^DATE^AND^TIME);
   L := 0;
   BASE := 10;
   WIDTH := 2;
   WHILE L < 6 DO
   BEGIN
      CALL NUMOUT(ORDER^RECORD.ORDER^NUMBER[L * 2],
                  J^DATE^AND^TIME[L],
                  BASE,WIDTH);
      L := L + 1;
   END;
   WIDTH := 3;
   CALL NUMOUT(ORDER^RECORD.ORDER^NUMBER[12],
               J^DATE^AND^TIME[6],
               BASE,WIDTH);
   CALL NUMOUT(ORDER^RECORD.ORDER^NUMBER[15],
               J^DATE^AND^TIME[7],
               BASE,WIDTH);

   ORDER^RECORD.ORDER^NUMBER[18] ':='
                     ORDER^REQUEST.PART^NUMBER FOR 10;

!  Copy customer information from order record into
!  order request:

   ORDER^RECORD.NAME.LAST ':=' ORDER^REQUEST FOR
                               ($LEN(ORDER^REQUEST) / 2);

!  Get the date ordered (today's date) and put it into the
!  order record:

   CALL TIMESTAMP(ORDER^RECORD.DATE^ORDERED);

!  Assign "NO" (new order) as the shipping status:

   ORDER^RECORD.SHIPPING^STATUS ':=' "NO";
```

```
!  Write the order record to the orders file:

   CALL WRITEX(ORD^FNUM,ORDER^RECORD,$LEN(ORDER^RECORD));
   IF <> THEN
   BEGIN
      CALL FILE_GETINFO_(ORD^FNUM,REPLY^ERROR);
      RETURN;
   END;

!  Complete the order reply:

   ORDER^REPLY.ORDER^NUMBER ':='
                           ORDER^RECORD.ORDER^NUMBER FOR 28;
   SBUFFER[0] ':=' ORDER^REPLY FOR $LEN(ORDER^REPLY);

   REPLY^LEN := $LEN(ORDER^REPLY);
   REPLY^ERROR := 0;
END;
```

```
!------------------------------------------------------------
! Procedure to process an Open system message (-103).  It
! places the process handle of the requester in the opener
! table, if there is room.  If the table is full, it
! rejects the open.
!------------------------------------------------------------

PROC PROCESS^OPEN^MESSAGE;

BEGIN
   INT I;
   INT J;
   INT COUNT;

!  Check if opener table full.  Return "file in use" error if
!  it is full:

   IF OPENER^TABLE.CURRENT^COUNT >= MAX^OPENERS THEN
   BEGIN
      REPLY^LEN := 0;
      REPLY^ERROR := 12;
      RETURN;
   END;

!  Put the process handle into the opener table at the first
!  empty location and increment the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF OPENER^TABLE.OCB[I].PROCESS^HANDLE[J] = -1
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN THEN
      BEGIN
         OPENER^TABLE.OCB[I] ':='
                RECEIVE^INFO[6] FOR ZSYS^VAL^PHANDLE^WLEN;
         OPENER^TABLE.OCB[I].FILE^NUMBER := RECEIVE^INFO[3];
         OPENER^TABLE.CURRENT^COUNT :=
                OPENER^TABLE.CURRENT^COUNT + 1;
         REPLY^LEN := 0;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to process a Close system message.  This
! procedure removes the requester from the opener table.
!-------------------------------------------------------------

PROC PROCESS^CLOSE^MESSAGE;

BEGIN
   INT   I;
   INT   J;
   INT   COUNT;

!  Check that the closing process is in the opener table.
!  If so, remove the entry from the opener table and
!  decrement the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN AND
         RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
      BEGIN
         OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                          ZSYS^VAL^PHANDLE^WLEN * [-1];
         OPENER^TABLE.CURRENT^COUNT :=
                          OPENER^TABLE.CURRENT^COUNT - 1;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;

!  If calling process not in opener table, return error 60:

   REPLY^ERROR := 60;
END;
```

```
!-----------------------------------------------------------
! Procedure to process a system message other than Open or
! Close.
!-------------------------------------------------------------

PROC PROCESS^OTHER^MESSAGE;

BEGIN

INT INDEX, STATUS, COUNT^READ;
INDEX := -1;
   DO BEGIN
      STATUS := OPENER_LOST_( BUFFER:COUNT^READ,
                OPENER^TABLE.OCB[1], INDEX,
                MAX^OPENERS, $LEN( OPENER^TABLE.OCB[1] ));
      IF STATUS = 6 THEN
         OPENER^TABLE.CURRENT^COUNT :=
                        OPENER^TABLE.CURRENT^COUNT - 1;
      END
   UNTIL STATUS = 0 OR STATUS = 2 OR STATUS 3 OR STATUS = 7;
   REPLY^ERROR := 0;
   REPLY^LEN := 0;

END;


!-------------------------------------------------------------
! Procedure to process a system message.
!-------------------------------------------------------------

PROC PROCESS^SYSTEM^MESSAGE;

BEGIN
   CASE BUFFER[0] OF
   BEGIN

      -103      -> CALL PROCESS^OPEN^MESSAGE;

      -104      -> CALL PROCESS^CLOSE^MESSAGE;

      OTHERWISE -> CALL PROCESS^OTHER^MESSAGE;
   END;
END;
```

```
!--------------------------------------------------------------
! Procedure to save the Startup message.
!--------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,
              MATCH) VARIABLE;
INT      .RUCB,
         .START^DATA,
         .MESSAGE,
          LENGTH,
          MATCH;

BEGIN

!  Copy the Startup message into the START^UP^MESSAGE
!  structure and save the message length:

   START^UP^MESSAGE.MSG^CODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;
```

```
!------------------------------------------------------------
! Procedure to perform initialization.  It calls INITIALIZER
! to read the Startup message then opens the IN file, the
! inventory file, and $RECEIVE, and then initializes the
! opener table.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING   .TERM^NAME[0:MAXFLEN - 1]; !terminal file name
   INT       TERMLEN;
   STRING   .RECV^NAME[0:MAXFLEN - 1]; !$RECEIVE file name
   STRING   .ORD^FNAME[0:MAXFLEN - 1]; !orders file name
   INT       ORD^FLEN;
   STRING   .INV^FNAME[0:MAXFLEN - 1]; !inventory file name
   INT       INV^FLEN;
   INT       RECV^DEPTH;               !receive depth
   INT       I;
   INT       ERROR;

!  Read the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Open the home terminal (IN file);

   ERROR := OLDFILENAME_TO_FILENAME_(START^UP^MESSAGE.INFILE,
                                     TERM^NAME:MAXFLEN,
                                     TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open $RECEIVE with a receive depth of 1 and to accept
!  system messages (the default):

   RECV^NAME ':=' "$RECEIVE" -> @S^PTR;
   RECV^DEPTH := 1;
   ERROR := FILE_OPEN_(RECV^NAME:@S^PTR '-' @RECV^NAME,
                       RECV^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       RECV^DEPTH);

!  Instruct the operating system to send status change messages
!  for processors in both local and remote systems.

   CALL MONITORCPUS( -1 );
   CALL MONITORNET( 1 );
```

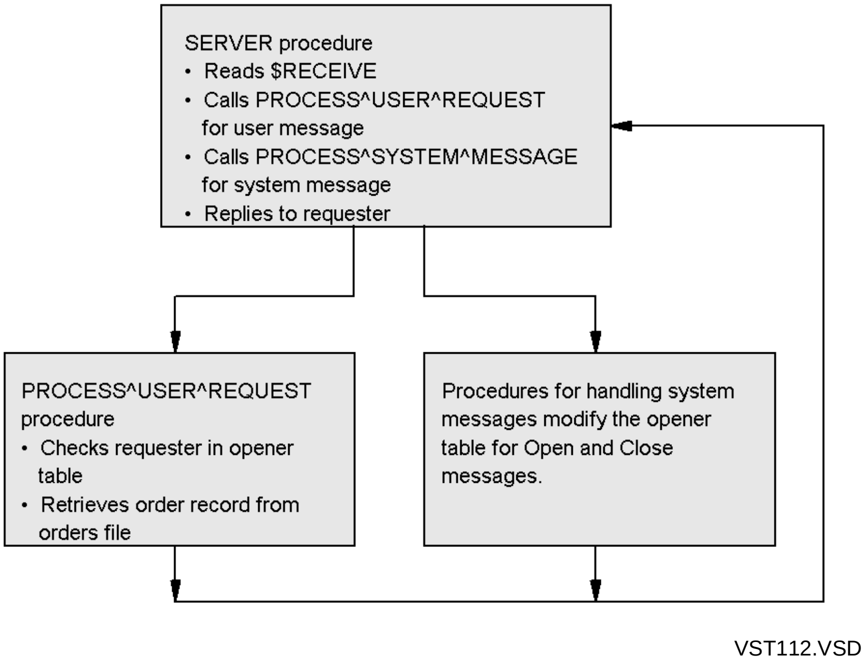
```
!  Open the orders file:

   ORD^FNAME ':=' "=ORD^FNAME" -> @S^PTR;
   ORD^FLEN := @S^PTR '-' @ORD^FNAME;
   ERROR := FILE_OPEN_(ORD^FNAME:ORD^FLEN,ORD^FNUM);
   IF ERROR <> 0 THEN
         CALL FILE^ERRORS^NAME(ORD^FNAME:ORD^FLEN,ERROR);

!  Open the inventory file:

   INV^FNAME ':=' "=INV^FNAME" -> @S^PTR;
   INV^FLEN := @S^PTR '-' @INV^FNAME;
   ERROR := FILE_OPEN_(INV^FNAME:INV^FLEN,INV^FNUM);
   IF ERROR <> 0 THEN
         CALL FILE^ERRORS^NAME(INV^FNAME:INV^FLEN,ERROR);

!  Initialize the opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                          [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      OPENER^TABLE.OCB[I].RESERVED^HANDLE ':='
                          [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      I := I + 1;
   END;
END;
```

```
!-------------------------------------------------------------
! Main procedure calls INIT to perform initialization and
! then goes into a loop in which it reads the $RECEIVE file.
! It calls the appropriate procedure depending on whether the
! message read was a system message, a user message, or
! whether the read operation generated an error.
!-------------------------------------------------------------

PROC SERVER MAIN;
BEGIN
   INT COUNT^READ;
   INT ERROR;

!  Initialize files and opener table:

   CALL INIT;

!  Loop forever:

   WHILE 1 DO
   BEGIN

   !  Read a message from $RECEIVE and check for an error:

      CALL READUPDATEX(RECV^NUM,SBUFFER,BUFSIZE,COUNT^READ);
      CALL FILE_GETINFO_(RECV^NUM,ERROR);

   !  Get the process handle of the requesting process:

      CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);

   !  Select a procedure depending on the results of the
   !  read operation:

      CASE ERROR OF
      BEGIN

      !  For a user message, call the PROCESS^USER^REQUEST
      !  procedure:

         0 -> CALL PROCESS^USER^REQUEST;

      !  For a system message, call the
      !  PROCESS^SYSTEM^MESSAGE procedure:

         6 -> CALL PROCESS^SYSTEM^MESSAGE;
         !  For any other error return, call the FILE^ERRORS
         !  procedure:

            OTHERWISE -> CALL FILE^ERRORS(RECV^NUM);
         END;

      !  Reply to the message:

         CALL REPLYX(SBUFFER,
                     REPLY^LEN,
                     !count^written!,
                     !message^tag!,
                     REPLY^ERROR);
      END;
   END;
```

The Order-Query Server ($SER3)

Figure 22-5 shows the function of each procedure in the order-query server and the relationships among the procedures.

Figure 22-5. Relationship Between Major Procedures in the Order-Query Server


SERVER procedure
• Reads $RECEIVE
• Calls PROCESS^USER^REQUEST for user message
• Calls PROCESS^SYSTEM^MESSAGE for system message
• Replies to requester
PROCESS^USER^REQUEST procedure
• Checks requester in opener table
• Retrieves order record from orders file
Procedures for handling system messages modify the opener table for Open and Close messages.
VST112.VSD

As you can see from Figure 22-5, the structure of the order-query server is similar to that of the part-query and process-order servers. The SERVER procedure and the procedures for handling system messages are the same as for the part-query and process-order servers. The differences are in the PROCESS^USER^REQUEST procedure.

The user message read by the SERVER procedure contains a 28-digit order number used to refer to the desired order record.

If the requester process is in the opener table, then the PROCESS^USER^REQUEST procedure uses the 28-digit order number provided in the message read by the SERVER procedure as a primary key to the orders file. After reading the record from the file, the procedure saves the record for the SERVER procedure to send back to the requester.

The Code for the Order-Query Server ($SER3)

The code for the order-query server program appears on the following pages.

```
?INSPECT, SYMBOLS, NOCODE
?NOLIST, SOURCE $TOOLS.ZTOOLD04.ZSYSTAL
?LIST

!-----------------------
!Literals:
!-----------------------

LITERAL MAX^OPENERS = 2,        !maximum number of openers
                                ! allowed
        EXACT       = 2,        !for exact key positioning
        MAXFLEN     = ZSYS^VAL^LEN^FILENAME,
                                !maximum length for file name
        BUFSIZE     = 512;

!------------------------
!Global data structures:
!------------------------

!Data structure for Startup message:

STRUCT .START^UP^MESSAGE;
BEGIN
   INT     MSG^CODE;
   STRUCT  DEFAULT;
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
   END;
   STRUCT  INFILE;              !IN file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
   STRUCT  OUTFILE;             !OUT file name
   BEGIN
      INT     VOLUME[0:3];
      INT     SUBVOLUME[0:3];
      INT     FILENAME[0:3];
   END;
   STRING  PARAM[0:529];        !parameter string
END;
INT    MESSAGE^LEN;             !length of Startup message

!Message received from requester.  Contains a part number:

STRUCT ORDER^QUERY;
BEGIN
   STRING ORDER^NUMBER[0:27];   !28-digit order number
END;


!Message returned to requester.  Contains order record
!information obtained from the orders file:

STRUCT .ORDER^REC;
BEGIN
   STRING ORDER^NUMBER[0:27];   !28-digit order number
   STRUCT  NAME;                !customer's name
   BEGIN
      STRING LAST[0:19];
      STRING FIRST[0:19];
      STRING INITIAL[0:1];
   END;
   STRING ADDRESS[0:47];        !customer's street address
   STRING CITY[0:23];           !city name
   STRING ZIP[0:7];             !customer's zip code
   STRING CCN[0:15];            !customer's credit-card
                                ! number
   STRING PART^NUMBER[0:9];     !part number of item ordered
   STRING PART^DESC[0:47];      !description of item ordered
   INT    QTY^ORDERED;          !quantity of item ordered
   INT    DATE^ORDERED[0:2];    !date that the order was
                                ! placed
   INT    SHIPPED[0:2];         !date order shipped to
                                ! customer
   STRING SHIPPING^STATUS[0:1]; !status of order
END;                            ! supplier


!Data structure for the opener table:

STRUCT .OPENER^TABLE;           !information about who has
BEGIN                           ! the server open
   INT CURRENT^COUNT;           !how many requesters have
                                ! this server open
   STRUCT OCB[1:MAX^OPENERS];   !one entry for each opener
   BEGIN

      !Process handle of an opener:

      INT PROCESS^HANDLE[0:9];    !process handle of opener
      INT RESERVED^HANDLE[0:9];   !reserved, filled with -1
      INT FILE^NUMBER;            !file number used by opener
      END;
   END;
```

```
!------------------------
!Other global variables:
!------------------------

STRING .S^PTR;                  !pointer to end of string

INT     TERM^NUM;               !file number for terminal
INT    .BUFFER[0:BUFSIZE/2 - 1];!I/O buffer
STRING .SBUFFER := @BUFFER[0] '<<' 1; !string pointer to I/O
                                ! buffer
INT     REPLY^LEN;              !length of reply string
INT     REPLY^ERROR;            !error value returned to
                                ! requester
INT     ORD^FNUM;               !file number for orders
                                ! file
INT     RECV^NUM;               !file number for $RECEIVE file
INT    .RECEIVE^INFO[0:16];     !returned by
                                ! FILE_GETRECEIVEINFO_

?NOLIST
?SOURCE
$SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,FILE_OPEN_,FILE_GETINFO_,
?                      PROCESS_STOP_,FILE_GETRECEIVEINFO_,
?                      KEYPOSITION,DNUMOUT,WRITEX,
?                      OLDFILENAME_TO_FILENAME_,READUPDATEX,
?                      REPLYX)
?LIST


!------------------------------------------------------------
!  Here are a few DEFINEs to make it a little easier to
!  format and print messages.
!------------------------------------------------------------

!  Initialize for a new line:

   DEFINE  START^LINE =    @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE  PUT^STR(S) =    S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N)  =
         @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;
```

```
!  Print the line:

   DEFINE  PRINT^LINE =
         CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE  PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

  DEFINE  PRINT^STR(S) =  BEGIN  START^LINE;
                                 PUT^STR(S);
                                 PRINT^LINE;   END;  #;


!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name and its length
! and the error number.  This procedure is used when the
! file is not open, so there is no file number for it.
!
! The procedure also stops the program after displaying the
! error message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING     .FNAME;
INT         LEN;
INT         ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error from $SER1 ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERM^NUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to display the
! information.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT   FNUM;
BEGIN
   INT     ERROR;
   STRING .FNAME[0:MAXFLEN-1];
   INT     FLEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,FLEN);
   CALL FILE^ERRORS^NAME(FNAME:FLEN,ERROR);
END;


!---------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, this procedure
! attempts to write a message about the error and then
! stops the program.
!---------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING   .BUF;
INT       LEN;
BEGIN
   CALL WRITEX(TERM^NUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERM^NUM);
END;
```

```
!-------------------------------------------------------------
! Procedure to process a request for an order record.  This
! procedure checks that the process that sent the message is
! in the opener table before retrieving the order record from
! the orders file using the key supplied in the order
! number.
!-------------------------------------------------------------

PROC PROCESS^USER^REQUEST;

BEGIN
   INT   POSITIONING^MODE;   !used by KEYPOSITION
   INT   COUNT^READ;
   INT   COUNT;
   INT   J;
   INT   I;

!  Check that the process handle of the requester is in the
!  opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN AND
          RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
      BEGIN

      !  Copy user message from requester into data
      !  structure:

         ORDER^QUERY.ORDER^NUMBER ':=' SBUFFER[0] FOR 28;

      !  Position pointers to appropriate record, based on
      !  the key value supplied in the request:

         POSITIONING^MODE := EXACT;
         CALL KEYPOSITION(ORD^FNUM,
                          ORDER^QUERY.ORDER^NUMBER,
                          !key^specifier!,
                          !length^word!,
                          POSITIONING^MODE);
         IF <> THEN CALL FILE^ERRORS(ORD^FNUM);
      !  Read the record from the orders file:

         CALL READUPDATEX(ORD^FNUM,SBUFFER,BUFSIZE,
                          COUNT^READ);

      !  If unable to position to the requested key, return
      !  the error number.  This error occurs when the key is
      !  not in the orders file.

         IF <> THEN
         BEGIN
            CALL FILE_GETINFO_(ORD^FNUM,REPLY^ERROR);
            REPLY^LEN := 0;
            RETURN;
         END;

      !  Clear the REPLY^ERROR variable and set the reply
      !  string length to the length of the orders
      !  record if the read is successful:

         REPLY^LEN := $LEN(ORDER^REC);
         REPLY^ERROR := 0;
         RETURN;
      END;

   !  Check next entry in the opener table:
      I := I + 1;
   END;

!  Requester not in opener table:
   REPLY^LEN := 0;
   REPLY^ERROR := 60;
END;
```

```
!-------------------------------------------------------------
! Procedure to process an Open system message (-103).  It
! places the process handle of the requester in the opener
! table, if there is room.  If the table is full, it
! rejects the open.
!-------------------------------------------------------------

PROC PROCESS^OPEN^MESSAGE;

BEGIN
   INT I;
   INT J;
   INT COUNT;

!  Check if opener table full.  Return "file in use" error if
!  it is full:

   IF OPENER^TABLE.CURRENT^COUNT >= MAX^OPENERS THEN
   BEGIN
      REPLY^LEN := 0;
      REPLY^ERROR := 12;
      RETURN;
   END;

!  Put the process handle into the opener table at the first
!  empty location and increment the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF OPENER^TABLE.OCB[I].PROCESS^HANDLE[J] = -1
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN THEN
      BEGIN
         OPENER^TABLE.OCB[I] ':='
                RECEIVE^INFO[6] FOR ZSYS^VAL^PHANDLE^WLEN;
         OPENER^TABLE.OCB[I].FILE^NUMBER := RECEIVE^INFO[3];
         OPENER^TABLE.CURRENT^COUNT :=
                OPENER^TABLE.CURRENT^COUNT + 1;
         REPLY^LEN := 0;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;
END;
```

```
!--------------------------------------------------------------
! Procedure to process a Close system message.  This
! procedure removes the requester from the opener table.
!--------------------------------------------------------------

PROC PROCESS^CLOSE^MESSAGE;

BEGIN
   INT   I;
   INT   J;
   INT   COUNT;

!  Check that the closing process is in the opener table.
!  If so, remove the entry from the opener table and
!  decrement the count of openers:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      J := 0;
      COUNT := 0;
      WHILE J <= (ZSYS^VAL^PHANDLE^WLEN - 1) DO
      BEGIN
         IF RECEIVE^INFO[J + 6] =
            OPENER^TABLE.OCB[I].PROCESS^HANDLE[J]
         THEN COUNT := COUNT + 1;
         J := J + 1;
      END;
      IF COUNT = ZSYS^VAL^PHANDLE^WLEN AND
         RECEIVE^INFO[3] = OPENER^TABLE.OCB[I].FILE^NUMBER THEN
      BEGIN
         OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                          ZSYS^VAL^PHANDLE^WLEN * [-1];
         OPENER^TABLE.CURRENT^COUNT :=
                          OPENER^TABLE.CURRENT^COUNT - 1;
         REPLY^ERROR := 0;
         RETURN;
      END;
      I := I + 1;
   END;

!  If calling process not in opener table, return error 60:

   REPLY^ERROR := 60;
END;
```

```
!--------------------------------------------------------------
! Procedure to process a system message other than Open or
! Close.
!--------------------------------------------------------------

PROC PROCESS^OTHER^MESSAGE;

BEGIN

INT INDEX, STATUS;
INDEX := -1;
   DO BEGIN
      STATUS := OPENER_LOST_( BUFFER:COUNT^READ,
                OPENER^TABLE.OCB[1], INDEX,
                MAX^OPENERS, $LEN( OPENER^TABLE.OCB[1] ));
      IF STATUS = 6 THEN
         OPENER^TABLE.CURRENT^COUNT :=
                        OPENER^TABLE.CURRENT^COUNT - 1;
      END
   UNTIL STATUS = 0 OR STATUS = 2 OR STATUS 3 OR STATUS = 7;
   REPLY^ERROR := 0;
   REPLY^LEN := 0;

END;


!--------------------------------------------------------------
! Procedure to process a system message.
!--------------------------------------------------------------

PROC PROCESS^SYSTEM^MESSAGE;

BEGIN
   CASE BUFFER[0] OF
   BEGIN

      -103      -> CALL PROCESS^OPEN^MESSAGE;

      -104      -> CALL PROCESS^CLOSE^MESSAGE;

      OTHERWISE -> CALL PROCESS^OTHER^MESSAGE;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to save the Startup message.
!-------------------------------------------------------------

PROC START^IT(RUCB,START^DATA,MESSAGE,LENGTH,
              MATCH) VARIABLE;
INT      .RUCB,
         .START^DATA,
         .MESSAGE,
          LENGTH,
          MATCH;

BEGIN

!  Copy the Startup message into the START^UP^MESSAGE
!  structure and save the message length:

   START^UP^MESSAGE.MSG^CODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;
```

```
!------------------------------------------------------------
! Procedure to perform initialization.  It calls INITIALIZER
! to read the Startup message then opens the IN file, the
! inventory file, and $RECEIVE, and then initializes the
! opener table.
!------------------------------------------------------------

PROC INIT;
BEGIN
   STRING   .TERM^NAME[0:MAXFLEN - 1]; !terminal file name
   INT       TERMLEN;
   STRING   .RECV^NAME[0:MAXFLEN - 1]; !$RECEIVE file name
   STRING   .ORD^FNAME[0:MAXFLEN - 1]; !data file name
   INT       ORD^FLEN;
   INT       RECV^DEPTH;               !receive depth
   INT       I;
   INT       ERROR;

!  Read the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    START^IT);

!  Open the home terminal (IN file);

   ERROR := OLDFILENAME_TO_FILENAME_(START^UP^MESSAGE.INFILE,
                                     TERM^NAME:MAXFLEN,
                                     TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

   ERROR := FILE_OPEN_(TERM^NAME:TERMLEN,TERM^NUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open $RECEIVE with a receive depth of 1 and to accept
!  system messages (the default):

   RECV^NAME ':=' "$RECEIVE" -> @S^PTR;
   RECV^DEPTH := 1;
   ERROR := FILE_OPEN_(RECV^NAME:@S^PTR '-' @RECV^NAME,
                       RECV^NUM,
                       !access!,
                       !exclusion!,
                       !nowait^depth!,
                       RECV^DEPTH);

!  Instruct the operating system to send status change messages
!  for processors in both local and remote systems.

   CALL MONITORCPUS( -1 );
   CALL MONITORNET( 1 );
```

```
!  Open the orders file:

   ORD^FNAME ':=' "=ORD^FNAME" -> @S^PTR;
   ORD^FLEN := @S^PTR '-' @ORD^FNAME;
   ERROR := FILE_OPEN_(ORD^FNAME:ORD^FLEN,ORD^FNUM);
   IF ERROR <> 0 THEN
         CALL FILE^ERRORS^NAME(ORD^FNAME:ORD^FLEN,ERROR);

!  Initialize the opener table:

   I := 1;
   WHILE I <= MAX^OPENERS DO
   BEGIN
      OPENER^TABLE.OCB[I].PROCESS^HANDLE ':='
                            [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      OPENER^TABLE.OCB[I].RESERVED^HANDLE ':='
                            [ ZSYS^VAL^PHANDLE^WLEN * [-1] ];
      I := I + 1;
   END;
END;
```

```
!------------------------------------------------------------
! Main procedure calls INIT to perform initialization and
! then goes into a loop in which it reads the $RECEIVE file.
! It calls the appropriate procedure depending on whether the
! message read was a system message, a user message, or
! whether the read operation generated an error.
!------------------------------------------------------------

PROC SERVER MAIN;
BEGIN
   INT COUNT^READ;
   INT ERROR;

!  Initialize files and opener table:

   CALL INIT;

!  Loop forever:

   WHILE 1 DO
   BEGIN

   !  Read a message from $RECEIVE and check for an error:

      CALL READUPDATEX(RECV^NUM,SBUFFER,BUFSIZE,COUNT^READ);
      CALL FILE_GETINFO_(RECV^NUM,ERROR);

   !  Get the process handle of the requesting process:

      CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);

   !  Select a procedure depending on the results of the
   !  read operation:

      CASE ERROR OF
      BEGIN

      !  For a user message, call the PROCESS^USER^REQUEST
      !  procedure:

         0 -> CALL PROCESS^USER^REQUEST;

      !  For a system message, call the
      !  PROCESS^SYSTEM^MESSAGE procedure:

         6 -> CALL PROCESS^SYSTEM^MESSAGE;

      !  For any other error return, call the FILE^ERRORS
      !  procedure:

         OTHERWISE -> CALL FILE^ERRORS(RECV^NUM);
      END;

   !  Reply to the message:
         CALL REPLYX(SBUFFER,
                     REPLY^LEN,
                     !count^written!,
                     !message^tag!,
                     REPLY^ERROR);
      END;
   END;
```

23

Writing a Command-Interpreter Monitor ($CMON)

A command-interpreter monitor process ($CMON) controls the operation of TACL processes. When a TACL process receives certain commands from a terminal user, it sends a request message to the $CMON process to have the request verified.

A $CMON process controls the following kinds of requests:

- Command-interpreter configuration requests
- Logon and logoff requests (LOGON and LOGOFF commands)
- Attempts to change user passwords (PASSWORD and REMOTEPASSWORD commands)
- Requests to create processes (implicit or explicit RUN commands)
- Requests to change process priority (ALTPRI command)
- Requests to add or delete users (ADDUSER and DELUSER commands)

The $CMON process receives requests from TACL processes by reading messages from its $RECEIVE file. $CMON processes each message and then sends a reply to the requesting TACL process. The $CMON process functions in the same way as any server process.

When replying to a command-interpreter request, $CMON can either accept the request, with or without modification, or reject the request and supply some display text giving the reason for the rejection.

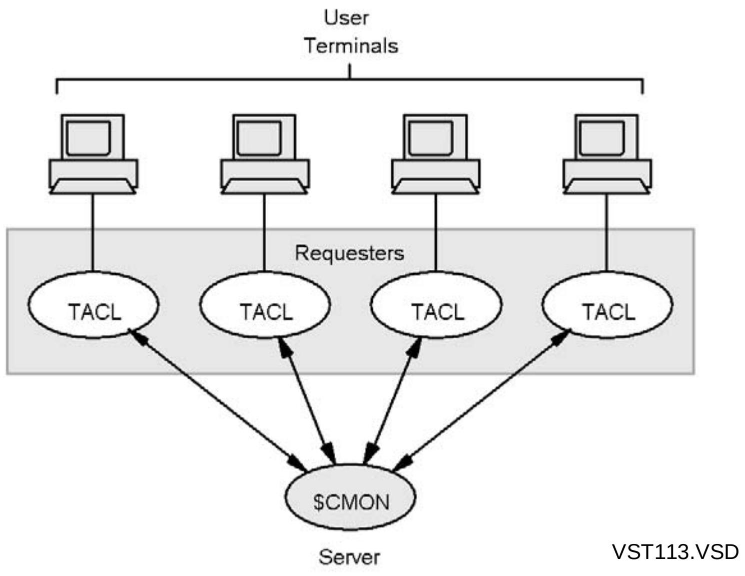
This section describes how you can write your own $CMON process. Your $CMON process can either provide static replies that are hard coded into the $CMON program or perform run-time control, allowing the operator to set reply information such as the text displayed at logon or the set of CPUs that a process is able to run in.

Although the examples given in this section are written in TAL, there is no need to use TAL to write a $CMON process. You can use any supported programming language: for example, COBOL, C, or Pascal.

Communicating With TACL Processes

Figure 23-1 shows the relationships between the $CMON process and the TACL processes.

Figure 23-1. Relationships Between TACL Processes and $CMON



Each TACL process opens the $CMON process the first time that it receives a command that causes a request to be sent to $CMON. If, for any reason, the TACL process is unable to open the $CMON process, then the command-interpreter process goes ahead and processes the command as if $CMON had accepted the request.

Once the TACL process has $CMON open, it communicates with $CMON as any requester process communicates with a server process. On receipt of an ALTPRI, LOGOFF, LOGON, or implicit or explicit RUN command, the TACL process sends the appropriate command-interpreter message to the $CMON process. The $CMON process reads the message from its $RECEIVE file using a call to the READUPDATE procedure. Section 6, Communicating With Processes, provides details of reading messages from $RECEIVE.

When the user issues an ADDUSER command, the ADDUSER process sends the ADDUSER message directly to the $CMON process (it does not come from the TACL process). Similarly, the DELUSER message is received from the DELUSER program, the PASSWORD message is received from the PASSWORD program, and the REMOTEPASSWORD message from the RPASSWRD program. For convenience, as well as historical consistency, these messages are still referred to as command-interpreter messages.

Note. Do not confuse command-interpreter messages with system messages; their message numbers do overlap. Command-interpreter messages are user messages supplied in a specific format. Unlike system messages, you do not get an error condition on reading a command-interpreter message.

After processing a command-interpreter message, $CMON sends a reply back to the TACL process (or other process that sent the message) using a call to the REPLY procedure. Each of the command-interpreter messages that contains a request has a specific reply format that the TACL process expects. Some of the reply messages have two possible formats; which one you choose usually depends on whether your $CMON process accepts or rejects the request. This section describes with examples the responses that a $CMON process can take for each of these messages. The *Guardian Procedure Errors and Messages Manual* also describes these messages, arranged numerically for ease of reference.

The following actions are also taken by the TACL process while communicating with the $CMON process:

- The TACL process sends all requests on a nowait basis; if a message cannot be sent, or if $CMON does not reply, the TACL process closes the $CMON file and proceeds to execute the command as if $CMON did not exist.
- If the BREAK key is pressed while a message is outstanding to $CMON and the user logged on to the TACL process is *not* the super ID (255, 255), the message is canceled and the command is aborted. If the BREAK key is pressed while a message is outstanding and the user logged on to the TACL process *is* the super ID, the message is canceled and the command is executed.
- If the TACL process encounters an I/O error when communicating with $CMON, it closes the $CMON file and abandons the current request. The TACL process tries to reopen $CMON and attempts communication when the next monitored command is issued.

Controlling the Configuration of a TACL Process

If a user is attempting to log on to an interactive TACL process from the logged-off state, or if a noninteractive TACL process is starting, the TACL process sends a Config^msg message to the $CMON process so it can verify or change the default parameters. If the REQUESTCMONUSERCONFIG configuration parameter is set to a nonzero value, the TACL process also sends a Config^msg message to the $CMON process after a logon is performed so it can obtain the user configuration and change the parameters accordingly; it does this after a logon from either the logged-off state or the logged-on state and after use of the #CHANGEUSER built-in function.

After processing a Config^msg message, the $CMON process replies with either a Config^text^reply message to keep the default configuration values as they are, or a Config^reply message, causing the TACL process to change the configuration parameters.

The Config^msg message has the message number -60 in its first word. The format of the message is given below:

Format of command-interpreter message -60 (Config^msg message):

```
STRUCT CONFIG^MSG;
BEGIN
   INT MSGCODE;              ![0]  value -60
   INT USERID;               ![1]  current user ID of TACL
                             !      process.
                             !      Value is 0 if logged off
                             !      or logged on as
                             !      NULL.NULL (0,0)
   INT CIPRI;                ![2]  current priority of TACL
                             !      process
   INT CIINFILE[0:11];       ![3]  IN file of TACL process
   INT CIOUTFILE[0:11];      ![15] OUT file of TACL process
   INT CONFIG_REQUEST_TYPE;  ![27] configuration request type
                             !      0 = send default
                             !          configuration
                             !      1 = send user
configuration
END;
```

Note. If the $CMON process intends to access the CONFIG_REQUEST_TYPE field of the Config^msg message, it should first check the length of the message (bytes read). Some processes, including earlier TACL versions, might not include this field with the message. In general, $CMON should be flexible about allowing callers to add new fields at the end of existing messages.

The configuration parameters are described below:

AUTOLOGOFFDELAY

> if greater than zero, specifies the maximum number of minutes that the TACL process is to wait at a prompt. If that time is exceeded, the TACL process automatically logs off.
>
> The default value is -1 (disabled).

BLINDLOGON

> if not zero, specifies that the LOGON command, whether in the logged-off state or the logged-on state, prohibits the use of the comma, requiring the password to be entered at its own prompt while echoing is disabled. The setting does not change the behavior of the #CHANGEUSER built-in function.
>
> The default value is 0.

CMONREQUIRED

> if not zero, specifies that all operations requiring approval by $CMON are denied if $CMON is not available or is running too slowly. Approval of $CMON is not required if the TACL process is already logged on as the super ID (255, 255).
>
> Use care when setting this flag. Note that if you set this flag on in the operator’s TACL process, the operator will not be able to log on to the system if $CMON is not running or is running slowly.
>
> The default value is 0.

CMONTIMEOUT

> specifies the number of seconds that the TACL process is to wait for any $CMON operation.
>
> The default value is 30.

LOGOFFSCREENCLEAR

> if not zero, specifies that if the TACL process is interactive and the IN file is a 65*xx* terminal, then the terminal is cleared at logoff unless the NOCLEAR option is supplied. The CLEAR and NOCLEAR options always override the automatic operation.
>
> The default value is -1.

NAMELOGON

> if not zero, specifies that the LOGON command, in both the logged-off and logged-on states, and the #CHANGEUSER built-in function do not accept user numbers but require that user names be used.
>
> The default value is 0.

NOCHANGEUSER

if zero, the #CHANGEUSER built-in function is enabled; users can log on from a terminal at which someone is already logged on. If -1, the #CHANGEUSER built-in function is disabled; users cannot log on from an already logged on terminal.

The default value is 0.

REMOTECMONREQUIRED

if not zero, specifies that all operations requiring approval by a remote $CMON are denied if that remote $CMON is unavailable or running too slowly. $CMON approval is not needed if the TACL process is already logged on as the super ID (255, 255).

The default value is 0.

REMOTECMONTIMEOUT

specifies the number of seconds that the TACL process is to wait for any $CMON operation involving a remote $CMON.

The default value is 30.

REMOTESUPERID

if zero, specifies that if the TACL process is started remotely, then any attempt to log on (or use the #CHANGEUSER built-in function) as the super ID (255, 255) results in an illegal logon.

The default value is -1 (disabled).

REQUESTCMONUSERCONFIG

if not zero, specifies that TACL send a Config^msg message to $CMON to retrieve the configuration data for the new user after a logon. (This is to be done after a logon from either the logged-off state or the logged-on state and after use of the #CHANGEUSER built-in function.) If zero, the Config^msg message is not sent to $CMON after logging on. (TACL sends a Config^msg message to $CMON before each logon from the logged-off state, or when a noninteractive TACL is starting, regardless of the value of this parameter.)

The default value is 0.

STOPONFEMODEMERR

if not zero, specifies that TACL stop when error 140 (FEMODEMERR) is encountered on its input. Control then returns to its parent process and the parent process receives a process termination message for the TACL process that stopped. If the TACL process was started with the PORTTACL startup parameter, this TACL configuration setting is ignored; in that case TACL goes to a logged off state and waits for a modem connect when error 140 is encountered.

The default value is 0 (TACL goes to a logged-off state and waits for a modem connect when error 140 is encountered).

Retaining Default Values

Your $CMON process can reply to the Config^msg message by returning a text message to the TACL process and accepting the default parameter values as they are. Here, $CMON returns the Config^text^reply structure as follows:

Format of Config^text^reply structure:

```
STRUCT CONFIG^TEXT^REPLY;
BEGIN
   INT REPLYCODE;             ![0] <> 0
   STRING REPLYTEXT[0:n];     ![1] display message; maximum
                              !     132 characters
END;
```

The following code fragment returns the Config^text^reply message without any text message. To do this, you simply set the length of the reply to two bytes and return a nonzero value in the reply code:

```
CONFIG^TEXT^REPLY.REPLY^CODE := -1;
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
BEGIN
   REPLY^LEN := 2;
   CALL REPLYX(CONFIG^TEXT^REPLY,REPLY^LEN);
END;
```

Setting Configuration Parameters

To change the configuration parameters, $CMON must reply with the Config^reply structure:

Format of Config^reply structure:

```
STRUCT CONFIG^REPLY;
BEGIN
   INT REPLYCODE;              ![0] value 0
   INT COUNT;                  ![1] number of integer
parameters
   INT AUTOLOGOFFDELAY;        ![2]
   INT LOGOFFSCREENCLEAR;      ![3]
   INT REMOTESUPERUSERID;      ![4]
   INT BLINDLOGON;             ![5]
   INT NAMELOGON;              ![6]
   INT CMONTIMEOUT;            ![7]
   INT CMONREQUIRED;           ![8]
   INT REMOTECMONTIMEOUT;      ![9]
   INT REMOTECMONREQUIRED;     ![10]
   INT NOCHANGEUSER;           ![11]
   INT STOPONFEMODEMERR;       ![12]
   INT REQUESTCMONUSERCONFIG; ![13]
END;
```

When you reply with the Config^reply message, you need to specify every value you return. For example, if you want to change only one or two values and leave the remaining values unchanged, you must specify the current values in the fields that remain unchanged. An exception is when replying with a short message: for example, if you want to change only the NAMELOGON parameter, then you could reply with a message that is just seven words long; the AUTOLOGOFFDELAY, LOGOFFSCREENCLEAR, REMOTESUPERUSERID, and BLINDLOGON parameters must contain the current values, but the CMONTIMEOUT, CMONREQUIRED, REMOTECMONTIMEOUT, REMOTECMONREQUIRED, NOCHANGEUSER, STOPONFEMODEMERR, and REQUESTCMONUSERCONFIG parameters need not be returned.

The following code fragment sets the values for AUTOLOGOFFDELAY (5 minutes) and CMONTIMEOUT (40 seconds):

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -60 THEN
BEGIN

!  Set the reply code to zero:

   CONFIG^REPLY.REPLYCODE := 0;

!  Set the new parameter values:

   CONFIG^REPLY.AUTLOGOFFDELAY := 5;
   CONFIG^REPLY.CMONTIMEOUT := 40;

!  Set default values for other parameters:

   CONFIG^REPLY.LOGOFFSCREENCLEAR := -1;
   CONFIG^REPLY.REMOTESUPERUSERID := -1;
   CONFIG^REPLY.BLINDLOGON := 0;
   CONFIG^REPLY.NAMELOGON := 0;
   CONFIG^REPLY.CMONREQUIRED := 0;
   CONFIG^REPLY.REMOTECMONTIMEOUT := 30;
   CONFIG^REPLY.REMOTECMONREQUIRED := 0;

   CONFIG^REPLY.COUNT := 9;

   CALL REPLYX(CONFIG^REPLY,$LEN(CONFIG^REPLY));
END;
```

Controlling Logon and Logoff

Your $CMON process can control attempts at logging on by accepting or rejecting logon attempts and by having messages displayed. $CMON can also cause a message to be displayed following either logoff or a failed attempt to log on.

The following paragraphs describe how to apply controls to attempts to log on, log off, and failed attempts to log on.

Controlling Logon

When a user attempts to log on to the system, the TACL process sends two messages to the $CMON process: the Prelogon^msg message and the Logon^msg message.

The Prelogon^msg Message

In addition to providing the user ID of the user logging on and information about the TACL process, the Prelogon^msg message provides the name of the user and information about whether the user is already logged on. Using this information, $CMON can invoke additional security, such as requiring a user to log on under one ID before logging on under another. The Prelogon^msg message has the following structure:

Structure of command-interpreter message -59 (Prelogon^msg message):

```
STRUCT PRELOGON^MSG;
BEGIN
   INT MSGCODE;              ![0]  value -59
   INT USERID;               ![1]  user ID of user logging on.
                             !      Value is 0 if user is
                             !      logged off or logged on as
                             !      NULL.NULL (0,0).
   INT CIPRI;                ![2]  current priority of command
                             !      interpreter
   INT CIINFILE[0:11];       ![3]  TACL process IN file
   INT CIOUTFILE[0:11];      ![15] TACL process OUT file
   INT LOGGEDON;             ![27] value is 0 if command
                             !      interpreter is currently
                             !      logged off, or nonzero if
                             !      TACL process is already
                             !      logged on
   INT USERNAME[0:7];        ![28] internal user name through
                             !      which the user wants to
                             !      log on
END;
```

After processing the Prelogon^msg message, the $CMON process replies with a Prelogon^reply structure in the format given below. The $CMON process has the option of accepting or rejecting the request and of sending a display message back to the TACL process. The display message is typically used to give a reason for the rejection.

Format of a Prelogon^reply structure:

```
STRUCT PRELOGON^REPLY;
BEGIN
   INT REPLYCODE;           ![0] if 0, proceed to VERIFYUSER;
                            !     if 1, disallow logon
   STRING REPLYTEXT[0:n];  ![1] optional display text;
                            !     maximum length is 132 bytes
END;
```

The following code fragment checks a flag value before deciding whether to accept the prelogon request. On rejection, this example returns the generic text “Prelogon rejected” to the TACL process; a typical response would indicate the reason for rejection:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -59 THEN
BEGIN
   IF ACCEPT^PRELOGON = YES THEN
   BEGIN
      PRELOGON^REPLY.REPLYCODE := 0;
      REPLY^LEN := 2;
   END
   ELSE
   BEGIN
      PRELOGON^REPLY.REPLYCODE := 1;
      PRELOGON^REPLY.REPLYTEXT ':=' ["Prelogon rejected",0];
      SCAN PRELOGON^REPLY.REPLYTEXT[0] UNTIL 0 -> @LAST;
      REPLY^LEN := 2 + @LAST - @PRELOGON^REPLY.REPLYTEXT[0];
   END;
   CALL REPLYX(PRELOGON^REPLY,REPLY^LEN);
END;
```

The Logon^msg Message

The Logon^msg message is sent to $CMON every time a user attempts to log on, giving $CMON the opportunity to accept or reject the logon. Note that this message does not contain information regarding whether the user is already logged on; that information was sent in the Prelogon^msg message. The Logon^msg message has the following structure:

Format of command-interpreter message -50 (Logon^msg message):

```
STRUCT LOGON^MSG;
BEGIN
   INT MSGCODE;               ![0]  value -50
   INT USERID;                ![1]  user ID of user logging on
   INT CIPRI;                 ![2]  current priority of command
                              !      interpreter
   INT CIINFILE[0:11];        ![3]  name of command file of
                              !      TACL process
   INT CIOUTFILE[0:11];       ![15] name of list file for the
                              !      TACL process
END;
```

After processing the Logon^msg message, the $CMON process replies with an indication of whether the user is allowed to log on. The reply can also contain display text. The structure of the reply message is as follows:

Format of Logon^reply structure:

```
STRUCT LOGON^REPLY;
BEGIN
   INT REPLYCODE;             ![0] if 0, proceed to VERIFYUSER;
                              !     if 1, disallow logon
   STRING REPLYTEXT[0:n];     ![1] optional display text;
                              !     maximum length is 132 bytes
END;
```

The following code fragment checks a flag value before deciding whether to accept the logon request:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -50 THEN
BEGIN
   IF ACCEPT^LOGON = YES THEN
   BEGIN
      LOGON^REPLY.REPLYCODE := 0;
      LOGON^REPLY.REPLYTEXT ':=' ["Welcome!",0];
   END;

   ELSE
   BEGIN
      LOGON^REPLY.REPLYCODE := 1;
      LOGON^REPLY.REPLYTEXT ':=' ["Logon rejected",0];
   END;
   SCAN LOGON^REPLY.REPLYTEXT[0] UNTIL 0 -> @LAST;
   REPLY^LEN := 2 + @LAST - @LOGON^REPLY.REPLYTEXT;
   CALL REPLYX(LOGON^REPLY,REPLY^LEN);
END;
```

Controlling Logoff

The TACL process sends a Logoff^msg message to $CMON whenever a user logs off either explicitly by issuing a LOGOFF command or implicitly by logging on without first logging off. This message gives $CMON the option of displaying message text on logging off. The $CMON process is not able to reject a request to log off.

The format of the Logoff^msg message is given below:

Format of command-interpreter message -51 (Logoff^msg message):

```
STRUCT LOGOFF^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -51
   INT USERID;             ![1]  user ID of user logging off
   INT CIPRI;              ![2]  current priority of
                           !      TACL process
   INT CIINFILE[0:11];     ![3]  name of the command file
                           !      for the TACL process
   INT CIOUTFILE[0:11];    ![15] name of the list file for
                           !      the TACL process
END;
```

The $CMON process replies using a Logoff^reply structure in the format shown below. Note that the reply code in this case is ignored by the TACL process because $CMON cannot reject a logoff request:

Format of Logoff^reply structure:

```
STRUCT LOGOFF^REPLY;
BEGIN
   INT REPLYCODE;        ![0]  ignored by TACL process
   STRING REPLYTEXT[0:n];![1]  optional display text;
                         !      maximum length is 132 bytes
END;
```

The following code fragment returns a message to the TACL process after receiving a logoff request:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -51 THEN
BEGIN
   LOGOFF^REPLY.REPLYTEXT ':=' ["Logging off...Bye!",0];
   SCAN LOGOFF^REPLY.REPLYTEXT UNTIL 0 -> @LAST;
   REPLY^LEN := 2 + @LAST - @LOGOFF^REPLY.REPLYTEXT;
   CALL REPLYX(LOGON^REPLY,REPLY^LEN);
END;
```

Controlling Illegal Logon

After the user has failed two consecutive times to log on to the system, the TACL process sends an Illegal^logon^msg message to $CMON on each subsequent failed logon attempt. The $CMON process replies with some message text to be displayed. $CMON cannot reject the Illegal^logon^msg message.

The Illegal^logon^msg message has the following structure:

Format of command-interpreter message -53 (Illegal^logon^msg message):

```
STRUCT ILLEGAL^LOGON^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -53
   INT USERID;             ![1]  user ID of user trying to
                           !      log on
   INT CIPRI;              ![2]  initial priority of command
                           !      interpreter
   INT CIINFILE[0:11];     ![3]  name of command file of
                           !      TACL process
   INT CIOUTFILE[0:11];    ![15] name of list file for the
                           !      TACL process
   INT LOGONSTRING[0:n];   ![27] the attempted LOGON command
                           !      string; maximum 132 bytes
END;
```

After processing the Illegal^logon^msg message, the $CMON process sends an Illegal^logon^reply structure back to the TACL process. The format of this message is as follows:

Format of Illegal^logon^reply structure:

```
STRUCT ILLEGAL^LOGON^REPLY;
BEGIN
   INT REPLYCODE;         ![0] ignored by TACL process
   STRING REPLYTEXT[0:n]; ![1] optional display text;
                          !     maximum length is 132 bytes
END;
```

The following code fragment returns a display message to the TACL process following receipt of an Illegal^logon^msg message:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -53 THEN
BEGIN
   ILLEGAL^LOGON^MSG ':=' BUFFER FOR BYTES^READ;
   ILLEGAL^LOGON^REPLY.REPLYTEXT ':='
                           "Invalid logon string: ";
   ILLEGAL^LOGON^REPLY.REPLYTEXT[11] ':='
                           [ILLEGAL^LOGON^MSG.LOGONSTRING FOR
                           BYTES^READ - 54,0];
   SCAN ILLEGAL^LOGON^REPLY.REPLYTEXT[0] UNTIL 0 -> @LAST;
   REPLY^LEN := 2 + @LAST - @ILLEGAL^LOGON^REPLY.REPLYTEXT;
   CALL REPLYX(LOGON^REPLY,REPLY^LEN);
END;
```

Controlling Passwords

A $CMON process can control the ability of a user to change passwords. This subsection describes how $CMON can provide this control for local passwords and remote passwords.

When the User Requests to Change a Local Password

When a user requests to change a local password by issuing a PASSWORD command, the PASSWORD process sends a Password^msg message to the $CMON process. The $CMON reply indicates whether the user’s password can be changed and contains optional display text.

The format of the Password^msg message is as follows:

Format of command-interpreter message -57 (Password^msg message):

```
STRUCT PASSWORD^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -57
   INT USERID;             ![1]  user ID of user requesting
                           !       a change of local password
   INT CIPRI;              ![2]  initial priority of command
                           !       interpreter
   INT CIINFILE[0:11];     ![3]  name of command file of
                           !       TACL process
   INT CIOUTFILE[0:11];    ![15] name of list file for the
                           !       TACL process
END;
```

After processing the Password^msg message, $CMON sends a Password^reply message back to the TACL process. The format of this structure is as follows:

Format of Password^reply structure:

```
STRUCT PASSWORD^REPLY;
BEGIN
   INT REPLYCODE;          ![0] if 0, allows the password
                           !     to be changed; if 1,
                           !     disallows a change of
                           !     password
   STRING REPLYTEXT[0:n]; ![1] optional display text;
                           !     maximum length is 132 bytes
END;
```

The following code fragment checks a flag value to see whether password changes are allowed, then returns a display message to the TACL process, depending on the setting of the flag:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -57 THEN
BEGIN
   IF CHANGE^PASSWORD = YES THEN
   BEGIN
      PASSWORD^REPLY.REPLYCODE := 0;
      PASSWORD^REPLY.REPLYTEXT ':='
                   ["Password change approved",0];
   END
   ELSE
   BEGIN
      PASSWORD^REPLY.REPLYCODE := 1;
      PASSWORD^REPLY.REPLYTEXT ':='
                   ["Password change rejected",0];
   END;
   SCAN PASSWORD^REPLY.REPLYTEXT[0] UNTIL 0 -> @LAST;
   REPLY^LEN := 2 + @LAST - @PASSWORD^REPLY.REPLYTEXT;
   CALL REPLYX(PASSWORD^REPLY,REPLY^LEN);
END;
```

When the User Requests to Change a Remote Password

When a user requests to change a remote password by issuing a REMOTEPASSWORD command, the RPASSWRD process sends a Remotepassword^msg message to the $CMON process. This message is like the Password^msg message, except that it also contains the name of the node on which the user wants to change the password. The $CMON reply indicates whether the user's password can be changed and contains optional display text.

The format of the Remotepassword^msg message is as follows:

Format of command-interpreter message -58 (Remotepassword^msg message):

```
STRUCT REMOTEPASSWORD^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -58
   INT USERID;             ![1]  user ID of user requesting
                           !      a change of remote password
   INT CIPRI;              ![2]  initial priority of command
                           !      interpreter
   INT CIINFILE[0:11];     ![3]  name of command file of
                           !      TACL process
   INT CIOUTFILE[0:11];    ![15] name of list file for the
                           !      TACL process
   INT SYSNAME[0:3];       ![27] change the remote password
                                  for this system
END;
```

After processing the Remotepassword^msg message, $CMON sends a Remotepassword^reply structure back to the TACL process. The format of this structure is as follows:

Format of Remotepassword^reply structure:

```
STRUCT REMOTEPASSWORD^REPLY;
BEGIN
   INT REPLYCODE;         ![0] if 0, allows the password
                          !     to be changed; if 1,
                          !     disallows a change of
                          !     password
   STRING REPLYTEXT[0:n]; ![1] optional display text;
                          !     maximum length is 132 bytes
END;
```

The following code fragment checks a flag value to see whether remote password changes are allowed, then returns display text to the TACL process, depending on the setting of the flag:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -58 THEN
BEGIN
   IF CHANGE^REMOTE^PASSWORD = YES THEN
   BEGIN
      REMOTEPASSWORD^REPLY.REPLYCODE := 0;
      REMOTEPASSWORD^REPLY.REPLYTEXT ':='
                   ["Password change approved",0];
   END
   ELSE
   BEGIN
      REMOTEPASSWORD^REPLY.REPLYCODE := 1;
      REMOTEPASSWORD^REPLY.REPLYTEXT ':='
                   ["Password change rejected",0];
   END;
   SCAN REMOTEPASSWORD^REPLY.REPLYTEXT[0] UNTIL 0 -> @LAST;
   REPLY^LEN := 2 + @LAST - @REMOTEPASSWORD^REPLY.REPLYTEXT;
   CALL REPLYX(REMOTEPASSWORD^REPLY,REPLY^LEN);
END;
```

Controlling Process Creation

When a user attempts to create a new process explicitly by issuing a RUN command, implicitly by typing an object-file name, or by using the #NEWPROCESS built-in function, the TACL process sends a Processcreation^msg message to the $CMON process to request verification of the process. The $CMON process may reject the request, or it may accept the request but optionally perform the following controls:

- Change the CPU in which the new process will execute
- Change the priority of the new process
- Change the program-file name from what was requested by the user

The structure of the Processcreation^msg message is as follows:

Format of command-interpreter message -52 (Processcreation^msg message):

```
STRUCT PROCESSCREATION^MSG;
BEGIN
   INT MSGCODE;             ![0]  value -52
   INT USERID;              ![1]  user ID or user logged on
   INT CIPRI;               ![2]  initial priority of command
                            !      interpreter
   INT CIINFILE[0:11];      ![3]  name of command file for
                            !      TACL process
   INT CIOUTFILE[0:11];     ![15] name of list file for
                            !      TACL process
   INT PROGNAME[0:11];      ![27] program-file name
   INT PRIORITY;            ![39] priority specified in RUN
                            !      command if supplied;
                            !      otherwise -1
   INT PROCESSOR;           ![40] processor number specified
in
                            !      RUN command if supplied;
                            !      otherwise -1
   INT PROGINFILE[0:11];    ![41] the expanded IN file RUN
                            !      parameter if supplied;
                            !      otherwise the default IN
                            !      file
   INT PROGOUTFILE[0:11];   ![53] the expanded OUT file RUN
                            !      parameter if supplied;
                            !      otherwise the default OUT
                            !      file
   INT PROGLIBFILE[0:11];   ![65] the expanded LIB file RUN
                            !      parameter if supplied;
                            !      otherwise blanks
   INT PROGSWAPFILE[0:11];  ![77] the expanded SWAP file RUN
                            !      parameter if supplied;
                            !      otherwise blanks
   INT PARAM_LEN;           ![89] length of PARAM in bytes.
   STRING PARAM[0:MAX^PARAM - 1];
                            ![90] parameter string of the RUN
                            !      command, which is up to
528
                            !      bytes in length including
2
                            !      null bytes at the end of
the
                            !      string.
END;
```

Note. The CIIN file name supplied in the process creation message is in local format even if the TACL process is remote. This means that the file name does not contain the node number. The $CMON process can get the node number of the CIIN file by calling LASTRECEIVE (or RECEIVEINFO) to get the process ID of the TACL that sent the process creation message; the process ID contains the node number of the TACL, which is the same as the node number of the CIIN file.

To allow process creation, $CMON returns the Processcreation^accept structure with the reply code set to 0. This structure also contains the name of the program file to run, the priority at which the process will run, and the number of the CPU in which the process will run. $CMON can change any of these values from those that it received in the Processcreation^msg message.

The format of the Processcreation^accept structure for allowing process creation is shown below:

Format of Processcreation^accept structure:

```
STRUCT PROCESSCREATION^ACCEPT;
BEGIN
   INT REPLY^CODE;          ![0]  0 to allow process creation
   INT PROGNAME[0:11];      ![1]  expanded name of program
                            !      file to be run
   INT PRIORITY;            ![13] execution priority of the
                            !      new process.
                            !      0 = PRI option specified
by
                            !          user. If no PRI option
                            !          is specified, then
TACL
                            !          priority minus 1.
                            !     >0 = execution priority
                            !     <0 = PRI option specified
by
                            !          user plus negative
                            !          priority offset
returned
                            !          in this field.
                            !          If no PRI option is
                            !          specified by user,
then
                            !          TACL priority minus 1
                            !          plus negative priority
                            !          offset returned in
this
                            !          field. For example,
                            !          PRI = 150,
                            !          priority = -5,
priority
                            !          used = 145).
   INT PROCESSOR;           ![14] processor where the new
                            !      process is to run or -1.
                            !      If -1, then the processor
                            !      in which the TACL process
                            !      is running is used.
END;
```

To reject a request to create a process, $CMON returns the Processcreation^reject message to the TACL process with the reply code set to 1 and the remainder of the message containing optional display text. The structure for rejecting a process-creation request is as follows:

Format of Processcreation^reject structure:

```
STRUCT PROCESSCREATION^REJECT;
BEGIN
   INT REPLYCODE;            ![0] 1 to reject process creation
   STRING REPLYTEXT[0:n];    ![1] optional message to be
                             !     displayed; maximum 132
                             !     bytes
END;
```

Controlling the Priority of a New Process

The PROCESSCREATION^MSG.PRIORITY variable in the request received by $CMON contains the priority requested by the user for the new process. If the user does not specify a priority, then this variable contains -1; a priority of -1 is interpreted as a priority of 1 less than that of the TACL process.

To accept the user's request for priority, $CMON either copies the requested (positive) priority into the PROCESSCREATION^ACCEPT.PRIORITY variable, or assigns a value of 0 to the variable (to indicate no change), and then sends the reply. To change the priority requested by the user, $CMON can either put the new value into the reply message, or it can put a negative value into the reply message; a negative value is added to the requested value, resulting in a reduced priority.

In the following example, the user's choice for priority is accepted unless the user requests a priority greater than 175. Here, $CMON reduces the priority to 175. The values for the program-file name and CPU number are returned unchanged:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -52 THEN
BEGIN
   PROCESSCREATION^MSG ':=' SBUFFER FOR
                            $LEN(PROCESSCREATION^MSG);
   IF PROCESSCREATION^MSG.PRIORITY > 175
   THEN PROCESSCREATION^ACCEPT.PRIORITY := 175
   ELSE PROCESSCREATION^ACCEPT.PRIORITY := 0;

   PROCESSCREATION^ACCEPT.PROGNAME ':='
        PROCESSCREATION^MSG.PROGNAME FOR 12;
   PROCESSCREATION^ACCEPT.PROCESSOR :=
        PROCESSCREATION^MSG.PROCESSOR;

   CALL REPLYX(PROCESSCREATION^ACCEPT,
              $LEN(PROCESSCREATION^ACCEPT));
END;
```

Controlling the CPU of a New Process

The user's choice of the CPU in which to run the new process is received by the $CMON process in the PROCESSCREATION^MSG.PROCESSOR variable. If the user did not specify a CPU, then this value contains -1.

The $CMON process can set the CPU number in the reply to any valid process number or -1 for the CPU in which the primary process of the TACL process is running.

The following example shows how $CMON can segregate processes into those that require priority response and those for which response time is not critical. Priority-response processes will run in CPUs 0, 2, and 4; other processes will run in CPUs 1, 3, and 5.

To keep the example simple, the requested priority is used to determine the speed of response required. A typical $CMON process might, for example, use a list of program names that run as priority-response processes. In this example, those processes with a requested priority greater than 150 are priority-response processes. Once the group of CPUs is established for a given process, the specific CPU is chosen on a round-robin basis.

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -52 THEN
BEGIN
   PROCESSCREATION^MSG ':=' SBUFFER FOR
                            $LEN(PROCESSCREATION^MSG);

!  Limit process priority to 175:

   IF PROCESSCREATION^MSG.PRIORITY > 175
   THEN PROCESSCREATION^ACCEPT.PRIORITY := 175
   ELSE PROCESSCREATION^ACCEPT.PRIORITY := 0;

!  Sort processes into priority and nonpriority response
!  and allocate to priority response and nonpriority
!  response processors:

   IF PROCESSCREATION^MSG.PRIORITY > 150
   THEN
   BEGIN
      PRIORITY^CPU := PRIORITY^CPU + 2;
      IF PRIORITY^CPU = 6 THEN PRIORITY^CPU := 0;
      PROCESSCREATION^ACCEPT.PROCESSOR := PRIORITY^CPU;
   END
   ELSE
   BEGIN
      NONPRIORITY^CPU := NONPRIORITY^CPU + 2;
      IF NONPRIORITY^CPU = 7 THEN NONPRIORITY^CPU := 1;
      PROCESSCREATION^ACCEPT.PROCESSOR := NONPRIORITY^CPU;
   END;
```

```
!  Do not change the program-file name:

   PROCESSCREATION^ACCEPT.PROGNAME ':='
        PROCESSCREATION^MSG.PROGNAME FOR 12;

!  Reply to the TACL process:

   CALL REPLYX(PROCESSCREATION^ACCEPT,
              $LEN(PROCESSCREATION^ACCEPT));
END;
```

Alternatively, you can prohibit use of certain CPUs that normally perform critical processing. Here, you could check the incoming request for a request to use the forbidden CPU and then either allocate some other CPU, again using a round-robin algorithm, or simply reject the request.

Controlling Change of Process Priority

When a user requests to change the priority of an existing process using the ALTPRI command, the TACL process sends an Altpri^msg message to $CMON to verify the request. The $CMON process can either accept the request as it is or reject it.

Note that for the user to change the priority of a process, one of the following must be true:

- The process has the same process access ID as the user.
- The user is the group manager of the process access ID of the process.
- The user is the super ID (255, 255).

If none of the above is true, then the user cannot change the priority, regardless of $CMON.

The format of the Altpri^msg message is as follows:

Format of command-interpreter message -56 (Altpri^msg message):

```
STRUCT ALTPRI^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -56
   INT USERID;             ![1]  user ID of user requesting
                           !      change of priority
   INT CIPRI;              ![2]  current priority of command
                           !      interpreter
   INT CIINFILE[0:11];     ![3]  name of command file of
                           !      TACL process
   INT CIOUTFILE[0:11];    ![15] name of list file for the
                           !      TACL process
   INT CRTPID[0:3];        ![27] process ID of process whose
                           !      priority is to be altered
   INT PROGNAME[0:11];     ![31] expanded program file name
of
                                  the process whose priority
is
                                  to be altered
   INT PRIORITY;           ![43] new priority
   INT PROCESS^HANDLE[0:9];![44] process handle of process
                           !      whose priority is to be
                           !      altered
END;
```

Note. The Altpri^msg message includes both the process ID (CRTPID field) and the process handle of the target process. The process ID field is retained for compatibility with C-series systems. A $CMON process running on a D-series system should always use the process-handle field to identify the target process.

The $CMON process must respond to an Altpri^msg message with an Altpri^reply structure; the reply code must be either 0 to allow the priority change or 1 to reject the priority change. The reply may also contain display text.

The format of the Altpri^reply structure is as follows:

Format of Altpri^reply structure:

```
STRUCT ALTPRI^REPLY;
BEGIN
   INT REPLYCODE;          ![0] if 0, allows the priority to
                          !     be changed; if 1, rejects the
                          !     attempt to change priority
   STRING REPLYTEXT[0:n]; ![1] optional display text;
                          !     maximum length is 132 bytes
END;
```

The following code fragment allows the priority to be reduced but not increased:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -56 THEN
BEGIN
   ALTPRI^MSG ':=' SBUFFER FOR BYTES^READ;
   ERROR := PROCESS_GETINFO_(ALTPRI^MSG.PROCESS^HANDLE,
                             !file^name:maxlen!,
                             !file^name^len!,
                             PRIORITY);
   IF ERROR <> 0 THEN
   BEGIN
      ALTPRI^REPLY.REPLYCODE = 1;
   END
   ELSE
   BEGIN
      IF PRIORITY > ALTPRI^MSG.PRIORITY
      THEN ALTPRI^REPLY.REPLYCODE = 0
      ELSE ALTPRI^REPLY.REPLYCODE = 1;
   END;
   REPLY^LEN := 2;
   CALL REPLYX(ALTPRI^REPLY,REPLY^LEN);
END;
```

Controlling Adding and Deleting Users

Attempts to add or delete users can be controlled by the $CMON process. The ADDUSER or DELUSER process asks $CMON for verification when a user issues an ADDUSER or DELUSER command.

Controlling Adding a User

When a user attempts to add a user to the system using the ADDUSER command, the ADDUSER process sends an Adduser^msg message to the $CMON process to verify the request. $CMON can either accept the request as it is or reject it.

Note that for a user to add a new user to the system, one of the following must be true:

- The user issuing the command is the group manager (*n*, 255) of the new user.
- The user issuing the command is the super ID (255, 255).

If neither of the above is true, then the user cannot add a new user, regardless of $CMON.

The format of the Adduser^msg message is as follows:

Format of command-interpreter message -54 (Adduser^msg message):

```
STRUCT ADDUSER^MSG;
BEGIN
   INT MSGCODE;            ![0]  value -54
   INT USERID;             ![1]  user ID of user making the
                           !      request
   INT CIPRI;              ![2]  initial priority of command
                           !      interpreter
   INT CIINFILE[0:11];     ![3]  name of command file of
                           !      TACL process
   INT CIOUTFILE[0:11];    ![15] name of list file for the
                           !      TACL process
   INT GROUPNAME[0:3];     ![27] the group name of the user
                           !      being added
   INT USERNAME[0:3];      ![31] name of the user being added
   INT GROUP^ID;           ![35] the group number of the user
                           !      being added
   INT USER^ID[0:3];       ![36] the user number of the user
                           !      being added
END;
```

The $CMON process must respond to an Adduser^msg message with an Adduser^reply structure; the reply code must be 0 to allow the new user or 1 to reject the new user. The reply may also contain display text.

The format of the Adduser^reply structure is as follows:

Format of Adduser^reply structure:

```
STRUCT ADDUSER^REPLY;
BEGIN
   INT REPLYCODE;         ![0] if 0, allows the user to
                          !    be added; if 1, rejects the
                          !    attempt to add a user
   STRING REPLYTEXT[0:n]; ![1] optional display text;
                          !     maximum length is 132 bytes
END;
```

The following code fragment rejects any attempt to add a user except by a super-group user (255, *n*):

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -54 THEN
BEGIN
   REQUESTING^GROUP^ID := ADDUSER^MSG.USERID.<0:7>;
   IF REQUESTING^GROUP^ID = 255
   THEN ADDUSER^REPLY.REPLY^CODE := 0
   ELSE ADDUSER^REPLY.REPLY^CODE := 1;
   REPLY^LEN := 2;
   CALL REPLYX(ADDUSER^REPLY,REPLY^LEN);
END;
```

Controlling Deleting a User

When a user attempts to delete another user from the system using the DELUSER command, the DELUSER process sends a Deluser^msg message to $CMON to verify the request. The $CMON process can either accept the request as it is or reject it.

Note that for the user to delete a user from the system, one of the following must be true:

- The user issuing the command is the group manager (*n*, 255) of the user to be deleted.
- The user issuing the command is the super ID (255, 255).

If neither of the above is true, then the requesting user cannot perform the deletion, regardless of $CMON.

The format of the Deluser^msg message is as follows:

Format of command-interpreter message -55 (Deluser^msg message):

```
STRUCT DELUSER^MSG;
BEGIN
   INT MSGCODE;              ![0]  value -55
   INT USERID;               ![1]  user ID of user requesting
                             !      to delete
   INT CIPRI;                ![2]  initial priority of command
                             !      interpreter
   INT CIINFILE[0:11];       ![3]  name of command file of
                             !      TACL process
   INT CIOUTFILE[0:11];      ![15] name of list file for the
                             !      TACL process
   INT GROUPNAME[0:3];       ![27] the group name of the user
                             !      being deleted
   INT USERNAME[0:3];        ![31] name of the user being
                             !      deleted
END;
```

The $CMON process must respond to a Deluser^msg message with a Deluser^reply structure; the reply code must be 0 to allow the deletion or 1 to reject the deletion. The reply may also contain display text.

The format of the Deluser^reply structure is as follows:

Format of Deluser^reply structure:

```
STRUCT DELUSER^REPLY;
BEGIN
   INT REPLYCODE;          ![0] if 0, allows the deletion;
                           !     if 1, rejects the deletion
   STRING REPLYTEXT[0:n];  ![1] optional display text;
                           !     maximum length is 132 bytes
END;
```

The following code fragment rejects any attempt to delete a user except by a super-group user (255, *n*):

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = -55 THEN
BEGIN
   REQUESTING^GROUP^ID := DELUSER^MSG.USERID.<0:7>;
   IF REQUESTING^GROUP^ID = 255
   THEN DELUSER^REPLY.REPLY^CODE := 0
   ELSE DELUSER^REPLY.REPLY^CODE := 1;
   REPLY^LEN := 2;
   CALL REPLYX(DELUSER^REPLY,REPLY^LEN);
END;
```

Controlling $CMON While the System Is Running

So far this section has discussed how $CMON provides static control over requests made by TACL process. However, you can write your $CMON program to permit run-time control over the way it responds to these messages. That is, instead of having responses hard coded into the $CMON process, the operator can supply or change response values at run time.

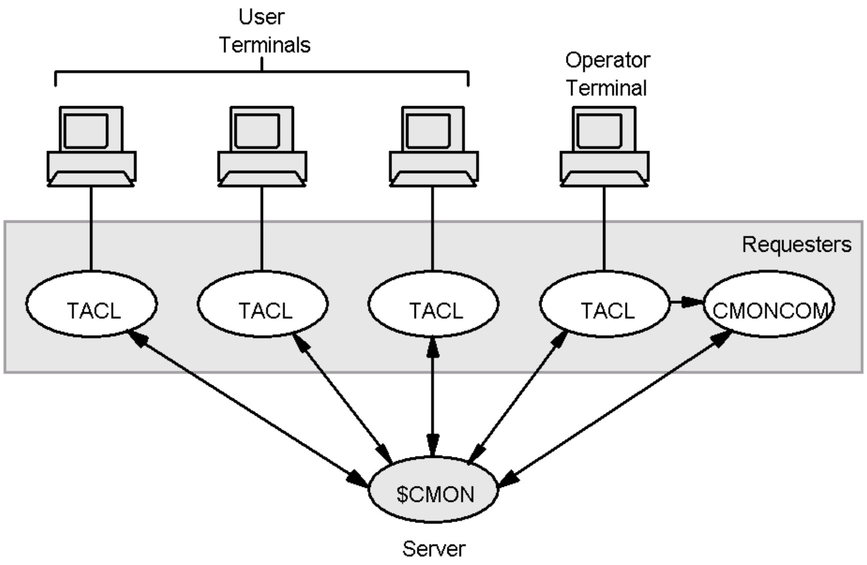
Any control that can be hard coded into $CMON can also be applied at run time. For example, your $CMON process might allow the operator to:

- Change the text displayed when logging on. For example, the operator may want to include some changing system-status information.
- Specify that only commands from operators are accepted. For example, the operator may want to forbid requests from users while preparing to shut down the system; requests from the operator, however, still need to be allowed.

- Control the CPU in which new processes can execute. This section has already described how to do this statically; your $CMON program can be written to allow CPU specification at run time.

One effective way of providing run-time control is to provide a command-interface program that shall be referred to as CMONCOM. This program is started by the operator's TACL process and passes requests to $CMON to set flags, provide response text, and so on. Figure 23-2 shows the model.

Figure 23-2. $CMON With Operator Control Process



VST114.VSD

For a model like this to work, you need a way to distinguish between messages from the command-interface program and messages received from the TACL process. Currently, all command-interpreter messages have a message code in the range -1 through -60. You should therefore choose message codes outside this range for messages from your command-interface program. Using positive numbers is one possible solution.

Note. HP reserves the right to add to its message-numbering system at any time. If you choose to use some currently unused message numbers for your own use, you should plan for the possibility that you might need to change them in the future.

Another check that your $CMON program should make is that the sender of a command-interface program message has proper authority. For example, you can check that the message sender belongs to the operations group.

Setting the Logon Display Text at Run Time

The following code fragments show how a command-interface program and a $CMON process can change the logon display text at run time. Here, (plus) 50 has been chosen as the message code for a message containing logon text.

The command-interface program prompts the operator for the logon text, puts the text into a data structure with the message code of 50, and then sends it to $CMON. $CMON reads the message from its $RECEIVE file, identifies it as a change logon text message, checks that the sender belongs to the operations group, and if so changes its logon-text buffer accordingly. $CMON will respond to future Logon^msg messages using the modified text string.

In the command-interface program:

```
STRUCT RT^LOGON^MSG;          !structure for run-time logon
BEGIN                         ! message
   INT MSGCODE;               !value (plus) 50
   STRING LOGON^MSG[0:n];     !the logon message; 132 bytes
END;                          ! maximum
 .
 .

!Prompt operator for new logon text:
SBUFFER ':=' "Enter New Logon Text" -> @S^PTR;
WCOUNT := @S^PTR '-' @SBUFFER;
RCOUNT := 132;
CALL WRITEREADX(TERM^NUM,SBUFFER,WCOUNT,RCOUNT,BYTES^READ);

!Fill in RT^LOGON^MSG data structure and send it to $CMON:
RT^LOGON^MSG.MSG^CODE := 50;
RT^LOGON^MSG.LOGON^MSG ':=' SBUFFER[0] FOR BYTES^READ;
WCOUNT := BYTES^READ + 2;
CALL WRITEREADX(CMON^NUM,RT^LOGON^MSG,WCOUNT,RCOUNT,
                BYTES^READ);
```

In the $CMON process:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = 50 THEN
BEGIN

!  Find out if sender is in group 255:

   CALL FILE_GETRECEIVEINFO_(INFO);
   P^HANDLE ':=' INFO[6] FOR 10;
   ERROR := PROCESS_GETINFO_(P^HANDLE,
                             !file^name:maxlen!,
                             !file^name^len!,
                             !priority!,
                             !moms^processhandle!,
                             !hometerm:maxlen!,
                             !hometerm^len!,
                             !process^time!,
                             !caid!,
                             PAID);
   IF PAID.<0:7> = 255 THEN

!  If sender is in group 255, process request

   BEGIN
      LOGON^MSG ':=' SBUFFER[2] FOR (BYTES^READ - 2);
      LOGON^MSG^LEN := BYTES^READ - 2;
   END;
   CALL REPLYX;
END;
```

Refusing Command-Interpreter Requests

You can write your $CMON and command-interface programs to set a flag that $CMON will check before replying to any command-interpreter message. If the flag is on, then $CMON rejects all requests. If the flag is off, then requests are processed normally.

To enable operator requests to continue to be accepted when requests from other users are rejected, $CMON should check the group number of the requesting process and allow the request if the group number is 255 (the group number for the operations group). If the group number is not 255, it should reject the request.

The following code fragments show the logic required in the control and $CMON processes to accomplish selective rejection of requests. The logic shown in $CMON is for the logon request but is the same for any other request.

In the command-interface program:

```
STRUCT RT^SHUTDOWN^MSG;        !structure for run-time logon
BEGIN                          ! message
   INT MSGCODE;                !value (plus) 61
   STRING SHUTDOWN^MSG[0:n];   !the reply text for subsequent
                               ! command-interpreter requests;
END;                           ! 132 bytes maximum
 .
 .

!Prompt operator for new logon text:
SBUFFER ':=' "Type 'x' to reject users: " -> @S^PTR;
WCOUNT := @S^PTR '-' @SBUFFER;
RCOUNT := 2;
CALL WRITEREADX(TERM^NUM,SBUFFER,WCOUNT,RCOUNT,BYTES^READ);

IF SBUFFER[0] = "x" THEN
BEGIN

!  Prompt operator for shutdown text:

   SBUFFER ':=' "Enter message text: " -> @S^PTR;;
   WCOUNT := S^PTR '-' @SBUFFER;
   CALL WRITEREADX(TERM^NUM,SBUFFER,WCOUNT,RCOUNT,
                   BYTES^READ);

!  Fill in RT^SHUTDOWN^MSG data structure and send it to
!  $CMON:

   RT^SHUTDOWN^MSG.MSG^CODE := 61;
   RT^SHUTDOWN^MSG.SHUTDOWN^MSG ':=' SBUFFER[0] FOR
                      BYTES^READ;
   WCOUNT := BYTES^READ + 2;
   CALL WRITEREADX(CMON^NUM,RT^SHUTDOWN^MSG,WCOUNT,RCOUNT,
                   BYTES^READ);
END;
```

In the $CMON process:

```
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = 61 THEN
BEGIN

!  Find out if sender is in group 255:

   CALL FILE_GETRECEIVEINFO_(INFO);
   P^HANDLE ':=' INFO[6] FOR 10;
   ERROR := PROCESS_GETINFO_(P^HANDLE,
                            !file^name:maxlen!,
                            !file^name^len!,
                            !priority!,
                            !moms^processhandle!,
                            !hometerm:maxlen!,
                            !hometerm^len!,
                            !process^time!,
                            !caid!,
                            PAID);
   IF PAID.<0:7> = 255 THEN

!  If request from operations group, process message:

   BEGIN
      REFUSE^ALL := YES;
      SHUTDOWN^TEXT ':=' SBUFFER[2] FOR (BYTES^READ - 2);
      SHUTDOWN^TEXT^LEN := BYTES^READ -2;
   END;
   CALL REPLYX;
END;
 .
 .

IF BUFFER[0] = -50 THEN            !logon request
BEGIN
   IF REFUSE^ALL = YES THEN
   BEGIN

   !  Reject if not operator group:

      REQUESTING^GROUP^ID := LOGON^MSG.USERID.<0:7>;
      IF REQUESTING^GROUP^ID <> 255 THEN
      BEGIN
         LOGON^REPLY.REPLYCODE := 1;
         LOGON^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR
                                    SHUTDOWN^TEXT^LEN;
      END
      ELSE
   !  Accept if operator group:

      BEGIN
         LOGON^REPLY.REPLYCODE := 0;
         LOGON^REPLY.REPLYTEXT ':=' LOGON^TEXT FOR
                                    LOGON^TEXT^LEN;
      END;
   END

!  Accept if REFUSE^ALL = NO:

   ELSE
   BEGIN
      LOGON^REPLY.REPLYCODE := 0;
      LOGON^REPLY.REPLYTEXT ':=' LOGON^TEXT FOR
                                 LOGON^TEXT^LEN;
   END;
   CALL REPLYX(LOGON^REPLY,$LEN(LOGON^REPLY));
END;
```

Controlling Which CPU a Process Can Run In

You can write your command-interface program to send information to the $CMON process indicating which CPU the process should run in. The following example expands the example given earlier for grouping processes according to their priority and having processes that need a priority response use one group of CPUs while those processes that are not so response-critical use the remaining CPUs. In this case, the operator is allowed to change the status of CPUs between priority response and non-priority response at run time.

This example is built around a table of CPUs in the $CMON process called the CPU^LIST. This is an integer array giving the response status for each CPU. CPU^LIST[0] represents CPU 0, and it is set to 1 if it is part of the priority-response group of CPUs or 0 if it belongs to the non-priority-response group of CPUs. CPU^LIST[1] represents CPU 1, and so on.

To change the status of a given CPU, the command-interface program formulates a message made up of the message code 62, the CPU number, and the new status. Then it sends the message to $CMON. On reading a message 62, $CMON updates the CPU^LIST accordingly.

When $CMON receives a Processcreation^msg message from a TACL process, it checks the priority and assigns the process to the priority or nonpriority group. $CMON then checks the CPU^LIST to find the next CPU allocated to the particular group. $CMON then puts that CPU number into the Processcreation^accept structure and returns the message to the TACL process.

In the command-interface program:

```
STRUCT CPU^CHANGESTATUS^MSG;
BEGIN
   INT MSGCODE;             !value 62
   INT PROCESSOR;           !processor number to change status
   INT STATUS;              !new status; 1 for move to
                            ! priority-response list, 0 for
                            ! move to non-priority-response
                            ! list
END;
 .
 .
!Set up the CPU^CHANGESTATUS^MSG message and send to $CMON:
CPU^CHANGESTATUS^MSG.MSGCODE := 62;
CPU^CHANGESTATUS^MSG.PROCESSOR := CPU^NUMBER;
CPU^CHANGESTATUS^MSG.STATUS := NEW^STATUS;
WCOUNT := $LEN(CPU^CHANGESTATUS^MSG);
CALL WRITEREADX(CMON^NUM,CPU^CHANGESTATUS^MSG,WCOUNT,RCOUNT,
                                               BYTES^READ);
```

In the $CMON process:

```
INT CPU^LIST[0:5];

CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
IF BUFFER[0] = 62 THEN
BEGIN

!  Find out if sender is in group 255:

   CALL FILE_GETRECEIVEINFO_(INFO);
   P^HANDLE ':=' INFO[6] FOR 10;
   ERROR := PROCESS_GETINFO_(P^HANDLE,
                             !file^name:maxlen!,
                             !file^name^len!,
                             !priority!,
                             !moms^processhandle!,
                             !hometerm:maxlen!,
                             !hometerm^len!,
                             !process^time!,
                             !caid!,
                             PAID);
   IF PAID.<0:7> = 255 THEN
END;

!  Process request if from operations group:

   CPU^LIST[BUFFER[1]] := BUFFER[2];
 .
 .
```

```
IF BUFFER[0] = -52 THEN
BEGIN
   PROCESSCREATION^MSG ':=' SBUFFER FOR
                            $LEN(PROCESSCREATION^MSG);

   !Limit process priority to 175:

   IF PROCESSCREATION^MSG.PRIORITY > 175
   THEN PROCESSCREATION^ACCEPT.PRIORITY := 175
   ELSE PROCESSCREATION^ACCEPT.PRIORITY := 0; ! Accept priority

!  Allocate priority-response processor if priority over 150,
!  otherwise allocate non-priority-response processor:

   IF PROCESSCREATION^MSG.PRIORITY > 150
   THEN
   BEGIN
      DO
      BEGIN
         PRIORITY^CPU := PRIORITY^CPU + 1;
         IF PRIORITY^CPU = 6 THEN PRIORITY^CPU := 0;
      END
      UNTIL CPU^LIST[PRIORITY^CPU] = 1;
      PROCESSCREATION^ACCEPT.PROCESSOR := PRIORITY^CPU;
   END
   ELSE
   BEGIN
      DO
      BEGIN
         NONPRIORITY^CPU := NONPRIORITY^CPU + 1;
         IF NONPRIORITY^CPU = 6 THEN NONPRIORITY^CPU := 0;
      END
      UNTIL CPU^LIST[NONPRIORITY^CPU] = 0;
      PROCESSCREATION^ACCEPT.PROCESSOR := NONPRIORITY^CPU;
   END;

!  Do not change the program-file name:

   PROCESSCREATION^ACCEPT.PROGNAME ':='
        PROCESSCREATION^MSG.PROGNAME FOR 12;

!  Reply to the TACL process:

   CALL REPLYX(PROCESSCREATION^ACCEPT,
              $LEN(PROCESSCREATION^ACCEPT));
END;
```

Writing a $CMON Program: An Example

The example presented here contains the code for two different processes:

- A $CMON process
- A command-interface program, CMONCOM

The $CMON process responds to all requests from a TACL process as well as to requests made by the command-interface program. The command-interface program makes requests for run-time control over how $CMON responds to requests made by a TACL process. Specifically, the command-interface program allows run-time control of:

- The logon text
- The logoff text
- Rejection of all requests that are not operator requests made while the system is shutting down
- Choice of CPU in which to run new processes

Note. This example is written in TAL. However, you could write the same programs using any supported programming language: for example, COBOL, C, or Pascal.

Sample $CMON Program

The $CMON program is made up the procedures described below. Apart from the main procedure, all $CMON procedures fall into two categories: procedures that respond to requests from the command-interface program and procedures that respond to requests from a TACL process.

Note. Each procedure that handles a command-interpreter request uses a structure template to gain access to fields of information within the received message. Although most of these procedures make little use of buffered information, writing the code this way makes it easier for future modification. A structure pointer maps the structure template to the I/O buffer.

- The CMON^MAIN procedure reads messages from the $RECEIVE file. Depending on the message code in the first word of each message, the CMON^MAIN procedure calls the specific procedure for processing that message. Positive message codes indicate messages from the command-interface program. Negative message codes indicate messages from a TACL process.
- The INIT and SAVE^STARTUP^MESSAGE procedures open the IN file and $RECEIVE and initialize some variables.
- The FILE^ERRORS and FILE^ERRORS^NAME procedures provide for file-system error handling. When a file-system error is encountered, these procedures attempt to display the name of the file that the error occurred against, as well as the file-system error number itself.

- The WRITE^LINE procedure provides a convenient way of writing a line to the terminal.

Procedures for Processing Requests From the Command-Interface Program

- The CHANGE^LOGON^MESSAGE procedure is called when $CMON receives an RT^LOGON^MESSAGE structure from the command-interface program. This procedure updates a buffer containing logon text using the text contained in the incoming message.
- The CHANGE^LOGOFF^MESSAGE procedure is called when $CMON receives an RT^LOGOFF^MESSAGE structure from the command-interface program. This procedure updates a buffer containing logoff text with the text contained in the incoming message.
- The REJECT^REQUESTS procedure is called when $CMON receives an RT^SHUTDOWN^MESSAGE structure from the command-interface program. The procedure sets the REFUSE^ALL flag on to prevent further command-interpreter requests from being accepted. In addition, the procedure updates a buffer containing shutdown text with the text string contained in the incoming message. This message is sent to TACL processes that make requests when the REFUSE^ALL flag is on.
- The ACCEPT^REQUESTS procedure is called when $CMON receives an RT^START^MESSAGE structure from the command-interface program. The procedure clears the REFUSE^ALL flag, allowing $CMON to accept command-interpreter requests.
- The CHANGE^CPU^STATUS procedure is called when $CMON receives a CPU^CHANGESTATUS^MESSAGE structure from the command-interface program. The incoming message contains a CPU number and CPU status. The procedure updates the CPU^LIST array to reflect the new CPU status.

Procedures for Processing Command-Interpreter Messages

- The PROCESS^CONFIG^MSG procedure is called when $CMON receives a Config^msg message from a TACL process. This procedure returns a blank text string to the TACL process; the TACL process retains the default logon parameters.
- The PROCESS^PRELOGON^MSG procedure is called when $CMON receives a Prelogon^msg message from a TACL process. This procedure normally accepts the request and returns a blank text string. The request is rejected for all nonoperator TACL processes only if the operator has already set the REFUSE^ALL flag to inhibit further requests.
- The PROCESS^LOGON^MSG procedure is called when $CMON receives a LOGON^MSG message from a TACL process. This procedure normally accepts the request and returns a text string for display. The request is rejected for all

nonoperator TACL processes only if the operator has already set the REFUSE^ALL flag to inhibit further requests.

- The PROCESS^LOGOFF^MSG procedure is called when $CMON receives a Logoff^msg message from a TACL process. It functions the same way as the PROCESS^LOGON^MSG procedure, except that it returns a logoff message instead of a logon message.
- The PROCESS^ILLEGAL^LOGON^MSG procedure is called when $CMON receives an Illegal^logon^msg message from a TACL process. It returns a blank text string to the TACL process.
- The PROCESS^PASSWORD^MSG procedure is called when $CMON receives a Password^msg message from the password program. This procedure normally accepts the request and returns a text string indicating that the password change is approved. The request is rejected for all nonoperator TACL processes only if the operator has already set the REFUSE^ALL flag to inhibit further requests.
- The PROCESS^REMOTEPASSWORD^MSG procedure is called when $CMON receives a Remotepassword^msg message from the RPASSWRD program. This procedure is just like the PROCESS^PASSWORD^MSG procedure, except that it works with remote passwords instead of local passwords.
- The PROCESS^PROCESSCREATION^MSG procedure is called when $CMON receives a Processcreation^msg message from a TACL process.

  This procedure normally accepts the program file and execution priority specified in the incoming message, but it assigns a CPU depending on the execution priority. Processes requested to run at priority 150 or higher are run in one set of CPUs, and processes with priority less than 150 run in the remaining CPUs. This procedure then assigns CPUs within each group on a round-robin basis. The procedure checks the CPU^LIST array to determine which group a CPU belongs to.

  The request is rejected for all nonoperator TACL processes only if the operator has already set the REFUSE^ALL flag to inhibit further requests.
- The PROCESS^ALTPRI^MSG procedure is called when $CMON receives an Altpri^msg message from a TACL process. This procedure changes the process priority only if the requester is trying to decrease the priority. The request is rejected for all nonoperator TACL processes if the operator has already set the REFUSE^ALL flag to inhibit further requests.
- The PROCESS^ADDUSER^MSG procedure is called when $CMON receives an Adduser^msg message from the ADDUSER program. This procedure accepts the request to add a user only if the originator of the request belongs to the operations group.
- The PROCESS^DELUSER^MSG procedure is called when $CMON receives a Deluser^msg message from the DELUSER program. This procedure accepts the request to delete a user from the system only if the originator of the request belongs to the operations group.

The $CMON Code

The code for the $CMON program follows.

```
?INSPECT, SYMBOLS, NOCODE, NOMAP
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!--------------------
!Literals:
!--------------------

LITERAL YES = 1;                 !setting for REFUSE^ALL flag
LITERAL NO = 0;                  !setting for REFUSE^ALL flag
LITERAL TOP^CPU^NUMBER = 5;      !highest CPU number
LITERAL BUFSIZE = 750;           !size of I/O buffer
LITERAL MAXFLEN = ZSYS^VAL^LEN^FILENAME; !maximum file-name
                                         ! length
LITERAL MAX^PARAM = 528;         !maximum length of process
                                 ! startup parameter string in
                                 ! startup message; this string
                                 ! can be passed in
                                 ! Processcreation^msg message.

!--------------------
!Global variables:
!--------------------

INT    .BUFFER[0:BUFSIZE/2 - 1]; !I/O buffer
STRING .SBUFFER :=
              @BUFFER[0] '<<' 1; !string pointer to I/O
                                 ! buffer
STRING .S^PTR;                   !string pointer

STRING .SHUTDOWN^TEXT[0:63] := "Shutting system down";
STRING .LOGON^TEXT[0:63]    := "Logon accepted";
STRING .LOGOFF^TEXT[0:63]   := "Logoff accepted";

INT     CPU^LIST[0:TOP^CPU^NUMBER]; !processor status array
INT     PRIORITY^CPU := 0;       !processor number of potential
                                 ! priority processor
INT     NONPRIORITY^CPU := 0;    !processor number of potential
                                 ! nonpriorty processor
INT REFUSE^ALL := NO;            !flag for rejecting/
                                 ! accepting command-
                                 ! interpreter requests
INT REQUESTING^GROUPID;          !group ID of command

INT TERMNUM;                     !terminal file number
INT RECVNUM;                     !$RECEIVE file number

STRUCT .CI^STARTUP;              !Startup message
BEGIN
   INT MSGCODE;
   STRUCT DEFAULTS;
```

```
      BEGIN
         INT VOLUME[0:3];
         INT SUBVOL[0:3];
      END;

      STRUCT INFILE;
      BEGIN
         INT VOLUME[0:3];
         INT SUBVOL[0:3];
         INT FILEID[0:3];
      END;
      STRUCT OUTFILE;
      BEGIN
         INT VOLUME[0:3];
         INT SUBVOL[0:3];
         INT FILEID[0:3];
      END;
   STRING PARAM[0:529];
   END;
   INT    MESSAGE^LEN;

   ?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
   ?  FILE_OPEN_,WRITEX,FILE_GETINFO_,READUPDATEX,WRITEREADX,
   ?  REPLYX,DNUMOUT,PROCESS_GETINFO_,PROCESS_STOP_,
   ?  OLDFILENAME_TO_FILENAME_)
   ?LIST

   !------------------------------------------------------------
   ! Here are some DEFINEs to help formatting and printing
   ! messages.
   !------------------------------------------------------------

   !  Initialize for a new line:

      DEFINE START^LINE = @S^PTR := @SBUFFER #;

   !  Put a string into the line:

      DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

   !  Put an integer into the line:

      DEFINE PUT^INT(N) =
             @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

   !  Print the line:

      DEFINE PRINT^LINE =
             CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

   !  Print a blank line:

      DEFINE PRINT^BLANK =
             CALL WRITE^LINE(SBUFFER,0); #;
```

```
!  Print a string:

   DEFINE PRINT^STR(S) = BEGIN  START^LINE;
                                PUT^STR(S);
                                PRINT^LINE; END #;

!-------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, length, and
! error number.  This procedure is mainly to be used when
! the file is not open, so there is no file number for it.
! FILE^ERRORS is to be used when the file is open.
!
! The procedure also stops the program after displaying the
! error message.
!-------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING   .FNAME;
INT       LEN;
INT       ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME for LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_;
END;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file
! name and error number are determined from the file number
! and FILE^ERRORS^NAME is then called to do the display.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT     FNUM;
BEGIN
   INT        ERROR;
   STRING    .FNAME[0:MAXFLEN - 1];
   INT        LEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,LEN);
   CALL FILE^ERRORS^NAME(FNAME:LEN,ERROR);
END;


!-------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, it attempts to write
! a message about the error and the program is stopped.
!-------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING     .BUF;
INT         LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to process the Config^msg message.  Accepts the
! current default values in all cases.
!------------------------------------------------------------

PROC PROCESS^CONFIG^MSG;
BEGIN
   STRUCT .CONFIG^TEXT^REPLY;
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Prepare the reply message:

   CONFIG^TEXT^REPLY.REPLYCODE := 1;
   CONFIG^TEXT^REPLY.REPLYTEXT[0] ':=' " ";
   CONFIG^TEXT^REPLY.REPLYTEXT[1] ':='
      CONFIG^TEXT^REPLY.REPLYTEXT[0] FOR 63;

!  Send the reply to the TACL process:

   CALL REPLYX(CONFIG^TEXT^REPLY,$LEN(CONFIG^TEXT^REPLY));
END;
```

```
!------------------------------------------------------------
! Procedure to process the Prelogon^msg message.  This
! request is accepted in all cases, except during the period
! before shutdown.
!------------------------------------------------------------

PROC PROCESS^PRELOGON^MSG;
BEGIN
   STRUCT MSG(*);              !template for PROCESS^PRELOGON
   BEGIN                       ! message
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT LOGGEDON;
      INT USERNAME[0:7];
   END;

   INT .EXT PRELOGON^MSG(MSG) := $XADR(BUFFER); !structure
                               ! pointer to I/O buffer

   STRUCT .PRELOGON^REPLY;     !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Clear text buffer for reply:

   PRELOGON^REPLY.REPLYTEXT[0] ':=' " ";
   PRELOGON^REPLY.REPLYTEXT[1] ':='
      PRELOGON^REPLY.REPLYTEXT[0] FOR 63;

!  Determine group ID of requester:

   REQUESTING^GROUPID := PRELOGON^MSG.USERID.<0:7>;

!  If shutting the system down, allow only operator group
!  requests:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      PRELOGON^REPLY.REPLYCODE := 1;
      PRELOGON^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END

!  Otherwise accept the request:

   ELSE PRELOGON^REPLY.REPLYCODE := 0;

!  Reply to TACL process:

   CALL REPLYX(PRELOGON^REPLY,$LEN(PRELOGON^REPLY));
END;
```

```
!------------------------------------------------------------
! Procedure to process a Logon^msg message.  The logon is
! always accepted, except after the shutdown phase has begun.
! The logon message returned to the TACL process can be
! changed at run time.
!------------------------------------------------------------

PROC PROCESS^LOGON^MSG;
BEGIN
   STRUCT MSG(*);             !template for LOGON^MSG message
   BEGIN
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
   END;

   INT .EXT LOGON^MSG(MSG) :=
        $XADR(BUFFER); !structure pointer to I/O buffer

   STRUCT .LOGON^REPLY;           !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the logon reply buffer:

   LOGON^REPLY.REPLYTEXT[0] ':=' " ";
   LOGON^REPLY.REPLYTEXT[1] ':=' LOGON^REPLY[0] FOR 63;

!  Extract the group ID of the requesting process:

   REQUESTING^GROUPID := LOGON^MSG.USERID.<0:7>;

!  If shutting the system down, accept only operator group
!  requests:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      LOGON^REPLY.REPLYCODE := 1;
      LOGON^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE
```

```
!  Otherwise accept the logon request and reply with the
!  logon text:

   BEGIN
      LOGON^REPLY.REPLYCODE := 0;
      LOGON^REPLY.REPLYTEXT ':=' LOGON^TEXT FOR 64;
   END;

!  Sends the reply message to the TACL process:

   CALL REPLYX(LOGON^REPLY,$LEN(LOGON^REPLY));
END;
```

```
!------------------------------------------------------------
! Procedure to process a Logoff^msg message.  The message is
! always accepted except during the shutdown phase.  The
! display text returned to the TACL process can be changed at
! run time.
!------------------------------------------------------------

PROC PROCESS^LOGOFF^MSG;
BEGIN
   STRUCT MSG(*);            !template for LOGOFF^MSG message
   BEGIN
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
   END;

   INT .EXT LOGOFF^MSG(MSG) := $XADR(BUFFER);  !structure
                             ! pointer to I/O buffer

   STRUCT .LOGOFF^REPLY;     !structure for the reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the reply text buffer:

   LOGOFF^REPLY.REPLYTEXT[0] ':=' " ";
   LOGOFF^REPLY.REPLYTEXT[1] ':='
      LOGOFF^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID of the requesting process:

   REQUESTING^GROUPID := LOGOFF^MSG.USERID.<0:7>;

!  If shutting the system down, allow only operator group
!  requests:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      LOGOFF^REPLY.REPLYCODE := 1;
      LOGOFF^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE

!  Otherwise accept the request and return the logoff text:

   BEGIN
      LOGOFF^REPLY.REPLYCODE := 0;
      LOGOFF^REPLY.REPLYTEXT ':=' LOGOFF^TEXT FOR 64;
   END;

!  Send the reply back to the TACL process:
   CALL REPLYX(LOGOFF^REPLY,$LEN(LOGOFF^REPLY));
END;


!-------------------------------------------------------------
! Procedure for processing an Illegal^logon^msg message.
! This message is always accepted, even during the shutdown
! phase.  This procedure simply returns blank display text to
! the TACL process.
!-------------------------------------------------------------

PROC  PROCESS^ILLEGAL^LOGON^MSG;
BEGIN
   STRUCT .ILLEGAL^LOGON^REPLY;     !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the reply buffer.  There is no need to set the reply
!  code because the TACL process ignores it:

   ILLEGAL^LOGON^REPLY.REPLYTEXT[0] ':=' " ";
   ILLEGAL^LOGON^REPLY.REPLYTEXT[1] ':='
      ILLEGAL^LOGON^REPLY.REPLYTEXT[0] FOR 63;

!  Send the reply to the TACL process:

   CALL REPLYX(ILLEGAL^LOGON^REPLY,
               $LEN(ILLEGAL^LOGON^REPLY));
END;
```

```
!-------------------------------------------------------------
! Procedure to process a Password^msg message.  This request
! is always accepted, except during the shutdown phase.
!-------------------------------------------------------------

PROC PROCESS^PASSWORD^MSG;
BEGIN
   STRUCT MSG(*);                 !template for Password^msg
   BEGIN                          ! message
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
   END;

   INT .EXT PASSWORD^MSG(MSG) :=
        $XADR(BUFFER);            !structure pointer to I/O
                                  ! buffer

   STRUCT .PASSWORD^REPLY;        !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the reply buffer:

   PASSWORD^REPLY.REPLYTEXT[0] ':=' " ";
   PASSWORD^REPLY.REPLYTEXT[1] ':='
      PASSWORD^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID of the requesting process:

   REQUESTING^GROUPID := PASSWORD^MSG.USERID.<0:7>;

!  If shutting the system down, accept only requests from
!  the operator group:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      PASSWORD^REPLY.REPLYCODE := 1;
      PASSWORD^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE
```

```
!  Otherwise accept the request and reply with display text:

   BEGIN
      PASSWORD^REPLY.REPLYCODE := 0;
      PASSWORD^REPLY.REPLYTEXT ':='
         "Password change approved";
   END;

!  Send reply to the TACL process:

   CALL REPLYX(PASSWORD^REPLY,$LEN(PASSWORD^REPLY));
END;
```

```
!-------------------------------------------------------------
! Procedure to process a Remotepassword^msg message.  This
! request is always accepted, except during the shutdown
! phase.
!-------------------------------------------------------------

PROC PROCESS^REMOTEPASSWORD^MSG;
BEGIN
   STRUCT MSG(*);                 !template for
   BEGIN                          ! Remotepassword^msg message
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT SYSNAME[0:3];
   END;

   INT .EXT REMOTEPASSWORD^MSG(MSG) :=
        $XADR(BUFFER);            !structure pointer to I/O
                                  ! buffer

   STRUCT .REMOTEPASSWORD^REPLY; !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the reply text:

   REMOTEPASSWORD^REPLY.REPLYTEXT[0] ':=' " ";
   REMOTEPASSWORD^REPLY.REPLYTEXT[1] ':='
      REMOTEPASSWORD^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID of the requesting TACL process:

   REQUESTING^GROUPID := REMOTEPASSWORD^MSG.USERID.<0:7>;

!  If shutting the system down, allow requests only from the
!  operator group:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      REMOTEPASSWORD^REPLY.REPLYCODE := 1;
      REMOTEPASSWORD^REPLY.REPLYTEXT ':='
         SHUTDOWN^TEXT FOR 64;
   END
   ELSE

!  Otherwise accept the request and return the display text:

   BEGIN
      REMOTEPASSWORD^REPLY.REPLYCODE := 0;
      REMOTEPASSWORD^REPLY.REPLYTEXT ':='
         "Password change approved";
   END;

!  Send the reply to the TACL process:

   CALL REPLYX(REMOTEPASSWORD^REPLY,
               $LEN(REMOTEPASSWORD^REPLY));
END;
```

```
!-------------------------------------------------------------
! Procedure to process a Processcreation^msg message.  This
! request is always accepted, except during the shutdown
! phase.  This procedure assigns the process to a processor
! according to whether the process runs at a high or low
! priority.  Processes with a higher priority run in one set
! of processors, whereas processes with priority 150 or less run
! in the remaining processors.  The allocation of processors to
! priority or nonpriority processes is run-time configurable;
! see the CHANGE^CPU^STATUS procedure.
!-------------------------------------------------------------

PROC PROCESS^PROCESSCREATION^MSG;
BEGIN
   STRUCT MSG(*);           !template for Processcreation^msg
   BEGIN                    ! message
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT PROGNAME[0:11];
      INT PRIORITY;
      INT PROCESSOR;
      INT PROGINFILE[0:11];
      INT PROGOUTFILE[0:11];
      INT PROGLIBFILE[0:11];
      INT PROGSWAPFILE[0:11];
      INT PARAM_LEN;
      STRING PARAM[0:MAX^PARAM - 1];
   END;

   INT .EXT PROCESSCREATION^MSG(MSG) :=
        $XADR(BUFFER);                !structure pointer
                                      ! to I/O buffer

   STRUCT .PROCESSCREATION^REJECT;    !structure for reject
   BEGIN                              ! reply
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

   STRUCT .PROCESSCREATION^ACCEPT;    !structure for accept
   BEGIN                              ! reply
      INT REPLYCODE;
      INT PROGNAME[0:11];
      INT PRIORITY;
      INT PROCESSOR;
   END;

   INT COUNT;
```

```
! Blank the reply buffer for rejecting requests:

  PROCESSCREATION^REJECT[0] ':=' " ";
  PROCESSCREATION^REJECT[1] ':='
     PROCESSCREATION^REJECT[0] FOR 63;

! Extract the group ID of the requesting TACL process:

  REQUESTING^GROUPID := PROCESSCREATION^MSG.USERID.<0:7>;

! If shutting the system down, allow only requests from the
! operator group:

  IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
  BEGIN
     PROCESSCREATION^REJECT.REPLYCODE := 1;
     PROCESSCREATION^REJECT.REPLYTEXT ':='
        SHUTDOWN^TEXT FOR 64;
     CALL REPLYX(PROCESSCREATION^REJECT,
                 $LEN(PROCESSCREATION^REJECT));
  END
  ELSE
  BEGIN

  ! Allow the request:

     PROCESSCREATION^ACCEPT.REPLYCODE := 0;

  ! Accept process priority indicated in the input message:

     PROCESSCREATION^ACCEPT.PRIORITY := 0;

  ! Allocate priority-response processor if priority over 150,
  ! otherwise allocate nonpriority-response processor:

     IF PROCESSCREATION^MSG.PRIORITY > 150
     THEN
     BEGIN
        DO
        BEGIN
           COUNT := 0;
           PRIORITY^CPU := PRIORITY^CPU + 1;
           IF PRIORITY^CPU = (TOP^CPU^NUMBER + 1)
              THEN PRIORITY^CPU := 0;
           COUNT := COUNT + 1;
           IF COUNT = 16 THEN
```

```
         !  There is no priority processor available, use a
         !  nonpriority-response processor:

            BEGIN
               NONPRIORITY^CPU := NONPRIORITY^CPU + 1;
               IF NONPRIORITY^CPU = (TOP^CPU^NUMBER + 1)
               THEN NONPRIORITY^CPU := 0;
               PROCESSCREATION^ACCEPT.PROCESSOR :=
                                      NONPRIORITY^CPU;
           END;

        !  Next priority processor found:

           IF CPU^LIST[PRIORITY^CPU] = 1 THEN
              PROCESSCREATION^ACCEPT.PROCESSOR :=
                                     PRIORITY^CPU;
         END
         UNTIL (CPU^LIST[PRIORITY^CPU] = 1 OR COUNT = 16);
      END
      ELSE
      BEGIN
         DO
         BEGIN
            COUNT := 0;
            NONPRIORITY^CPU := NONPRIORITY^CPU + 1;
            IF NONPRIORITY^CPU = (TOP^CPU^NUMBER +1)
               THEN NONPRIORITY^CPU := 0;
            COUNT := COUNT + 1;
            IF COUNT = 16 THEN

         !  There is no nonpriority processor available, use a
         !  priority processor:

            BEGIN
               PRIORITY^CPU := PRIORITY^CPU + 1;
               IF PRIORITY^CPU = (TOP^CPU^NUMBER + 1)
               THEN PRIORITY^CPU := 0;
               PROCESSCREATION^ACCEPT.PROCESSOR :=
                                      PRIORITY^CPU;
            END;

         !  Next nonpriority processor found:

            IF CPU^LIST[PRIORITY^CPU] = 1 THEN
               PROCESSCREATION^ACCEPT.PROCESSOR :=
                                      NONPRIORITY^CPU;
         END
         UNTIL (CPU^LIST[NONPRIORITY^CPU] = 0 OR COUNT = 16);
      END;
      !  Do not change the program-file name:

         PROCESSCREATION^ACCEPT.PROGNAME ':='
            PROCESSCREATION^MSG.PROGNAME FOR 11;

      !  Reply to the TACL process:

         CALL REPLYX(PROCESSCREATION^ACCEPT,
                     $LEN(PROCESSCREATION^ACCEPT));
      END;
   END;
```

```
!--------------------------------------------------------------
! Procedure to process an Altpri^msg message.  This request
! is rejected during the shutdown phase.  Otherwise, the
! request is accepted only if it reduces the priority of
! the process.
!--------------------------------------------------------------

PROC PROCESS^ALTPRI^MSG;
BEGIN
   STRUCT MSG(*);            !template for Altpri^msg message
   BEGIN
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT CRTPID[0:3];
      INT PROGNAME[0:11];
      INT PRIORITY;
      INT PHANDLE[0:9];
   END;

   INT .EXT ALTPRI^MSG(MSG) :=
        $XADR(BUFFER);       !structure pointer to I/O buffer

   STRUCT .ALTPRI^REPLY;     !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

   INT PRIORITY;             !current priority of process

!  Blank the reply display text:

   ALTPRI^REPLY.REPLYTEXT[0] ':=' " ";
   ALTPRI^REPLY.REPLYTEXT[1] ':='
      ALTPRI^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID:

   REQUESTING^GROUPID := ALTPRI^MSG.USERID.<0:7>;

!  If shutting the system down, accept requests only from
!  the operator group:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      ALTPRI^REPLY.REPLYCODE := 1;
      ALTPRI^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE
```

```
!  If accepting the request:

   BEGIN

   !  Allow priority change only if operator group attempting
   !  to reduce priority:

      CALL PROCESS_GETINFO_(ALTPRI^MSG.PHANDLE,
                            !file^name:maxlen!,
                            !file^name^len!,
                            PRIORITY);
      IF (PRIORITY > ALTPRI^MSG.PRIORITY) OR
         (REQUESTING^GROUPID = 255) THEN
      ALTPRI^REPLY.REPLYCODE := 0
      ELSE
      BEGIN
         ALTPRI^REPLY.REPLYCODE := 1;
         ALTPRI^REPLY.REPLYTEXT ':='
            "Cannot increase process priority";
      END;
   END;

!  Send the reply to the TACL process:

   CALL REPLYX(ALTPRI^REPLY,$LEN(ALTPRI^REPLY));
END;
```

```
!------------------------------------------------------------
! Procedure to process an Adduser^msg message.  This request
! is rejected during the shutdown phase.  The request is
! accepted only if the request comes from a super-group
! user (255, n).
!------------------------------------------------------------

PROC PROCESS^ADDUSER^MSG;
BEGIN
   STRUCT MSG(*);           !template for Adduser^msg message
   BEGIN
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT GROUPNAME[0:3];
      INT USERNAME[0:3];
      INT GROUPID;
      INT USER^ID;
   END;

   INT .EXT ADDUSER^MSG(MSG) :=
        $XADR(BUFFER);      !structure pointer to I/O buffer

   STRUCT .ADDUSER^REPLY;   !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the reply display-text buffer:

   ADDUSER^REPLY.REPLYTEXT[0] ':=' " ";
   ADDUSER^REPLY.REPLYTEXT[1] ':='
      ADDUSER^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID;

   REQUESTING^GROUPID := ADDUSER^MSG.USERID.<0:7>;

!  If shutting the system down, accept the request only if
!  from the operator group:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      ADDUSER^REPLY.REPLYCODE := 1;
      ADDUSER^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE

!  Otherwise, accept request only if originating from
!  operator group.

   BEGIN
      IF REQUESTING^GROUPID = 255 THEN
      ADDUSER^REPLY.REPLYCODE := 0
      ELSE
      BEGIN
         ADDUSER^REPLY.REPLYCODE := 1;
         ADDUSER^REPLY.REPLYTEXT ':=' "Must be operator";
      END;
   END;

!  Send reply to TACL process:

   CALL REPLYX(ADDUSER^REPLY,$LEN(ADDUSER^REPLY));
END;
```

```
!------------------------------------------------------------
! Procedure to process a Deluser^msg message.  This request
! is rejected during the shutdown phase.  The request is
! accepted only if the request comes from a super-group
! user (255, n).
!------------------------------------------------------------

PROC PROCESS^DELUSER^MSG;
BEGIN
   STRUCT MSG(*);           !template for Deluser^msg message
   BEGIN
      INT MSGCODE;
      INT USERID;
      INT CIPRI;
      INT CIINFILE[0:11];
      INT CIOUTFILE[0:11];
      INT GROUPNAME[0:3];
      INT USERNAME[0:3];
   END;

   INT .EXT DELUSER^MSG(MSG) :=
        $XADR(BUFFER);      !structure pointer to I/O buffer

   STRUCT .DELUSER^REPLY;   !structure for reply
   BEGIN
      INT REPLYCODE;
      STRING REPLYTEXT[0:63];
   END;

!  Blank the display text buffer for the reply:

   DELUSER^REPLY.REPLYTEXT[0] ':=' " ";
   DELUSER^REPLY.REPLYTEXT[1] ':='
      DELUSER^REPLY.REPLYTEXT[0] FOR 63;

!  Extract the group ID of the requesting TACL process:

   REQUESTING^GROUPID := DELUSER^MSG.USERID.<0:7>;

!  If shutting the system down, accept requests only from
!  the operator group:

   IF REFUSE^ALL AND (REQUESTING^GROUPID <> 255) THEN
   BEGIN
      DELUSER^REPLY.REPLYCODE := 1;
      DELUSER^REPLY.REPLYTEXT ':=' SHUTDOWN^TEXT FOR 64;
   END
   ELSE

!  Otherwise, accept the request only if from operator group:

   BEGIN
      IF REQUESTING^GROUPID = 255 THEN
      DELUSER^REPLY.REPLYCODE := 0
      ELSE
      BEGIN
         DELUSER^REPLY.REPLYCODE := 1;
         DELUSER^REPLY.REPLYTEXT ':=' "Must be operator";
      END;
   END;

!  Send reply to TACL process:

   CALL REPLYX(DELUSER^REPLY,$LEN(DELUSER^REPLY));
END;


!------------------------------------------------------------
! Procedure to process a Change^logon^message.  This
! procedure takes the logon display text from the input
! message and puts it in the LOGON^TEXT array to be read by
! the PROCESS^LOGON^MSG procedure when a user tries to log
! on.
!------------------------------------------------------------

PROC CHANGE^LOGON^MESSAGE;
BEGIN
   STRUCT MSG(*);            !template for Change^logon^message
   BEGIN
      INT MSGCODE;
      STRING DISPLAYTEXT[0:63];
   END;

   INT .EXT RT^LOGON^MESSAGE(MSG) :=
        $XADR(BUFFER);       !structure pointer to I/O buffer

!  Set the logon display text in the reply:

   LOGON^TEXT ':=' RT^LOGON^MESSAGE.DISPLAYTEXT FOR 64;
   CALL REPLYX;
END;
```

```
!------------------------------------------------------------
! Procedure to process a Change^logoff^message.  This
! procedure takes the logoff display text from the input
! message and puts it in the LOGOFF^TEXT array to be read by
! the PROCESS^LOGOFF^MSG procedure when a user tries to log
! off.
!------------------------------------------------------------

PROC CHANGE^LOGOFF^MESSAGE;
BEGIN
   STRUCT MSG(*);         !template for Change^logoff^message
   BEGIN
      INT MSGCODE;
      STRING DISPLAYTEXT[0:63];
   END;

   INT .EXT RT^LOGOFF^MESSAGE(MSG) :=
        $XADR(BUFFER);    !structure pointer to I/O buffer

!  Set the logoff display text in the reply:

   LOGOFF^TEXT ':=' RT^LOGOFF^MESSAGE.DISPLAYTEXT FOR 64;
   CALL REPLYX;
END;
```

```
!-------------------------------------------------------------
! Procedure responds to an RT^shutdown^message.  This
! procedure sets a flag that prohibits  $CMON from accepting
! any further requests from nonoperator TACL
! processes.
!-------------------------------------------------------------

PROC REJECT^REQUESTS;
BEGIN
   STRUCT MSG(*);           !template for RT^shutdown^message
   BEGIN
      INT MSGCODE;
      STRING SHUTDOWNTEXT[0:63];
   END;

   INT .EXT RT^SHUTDOWN^MESSAGE(MSG) :=
        $XADR(BUFFER);      !structure pointer to I/O buffer

!  Set the REFUSE^ALL flag:

   REFUSE^ALL := YES;

!  Set the shutdown display text in the reply:

   SHUTDOWN^TEXT ':='
      RT^SHUTDOWN^MESSAGE.SHUTDOWNTEXT FOR 64;
   CALL REPLYX;
END;


!-------------------------------------------------------------
! Procedure responds to an RT^startup^message.  This
! procedure sets a flag that reenables $CMON to accept
! requests from nonoperator TACL processes.
!-------------------------------------------------------------

PROC ACCEPT^REQUESTS;
BEGIN

   REFUSE^ALL := NO;
   CALL REPLYX;
END;
```

```
!------------------------------------------------------------
! Procedure to respond to a CPU^changestatus^message.  This
! procedure extracts a processor number and status from the
! incoming message and updates the status of the processor
! accordingly in the CPU^LIST array.
!------------------------------------------------------------

PROC CHANGE^CPU^STATUS;
BEGIN
   STRUCT MSG(*);      !template for CPU^changestatus^message
   BEGIN
      INT MSGCODE;
      INT PROCESSOR;
      INT STATUS;
   END;

   INT .EXT CPU^CHANGESTATUS^MESSAGE(MSG) :=
        $XADR(BUFFER); !structure pointer to I/O buffer

!  Set the new processor status in the reply:

   CPU^LIST[CPU^CHANGESTATUS^MESSAGE.PROCESSOR] :=
                    CPU^CHANGESTATUS^MESSAGE.STATUS;
   CALL REPLYX;
END;


!------------------------------------------------------------
! Procedure to respond to an unexpected message.  This
! procedure returns error 2 (invalid operation) to the TACL
! process.
!------------------------------------------------------------

PROC UNEXPECTED^MESSAGE;
BEGIN
   INT ERROR;

   ERROR := 2;
   CALL REPLYX(!buffer!,
               !write^count!,
               !count^written!,
               !message^tag!,
               ERROR);
END;
```

```
!------------------------------------------------------------
! Procedure to save the Startup message in the global data
! area.
!------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH,MATCH)VARIABLE;

INT  .RUCB;
INT  .START^DATA;
INT  .MESSAGE;
INT   LENGTH;
INT   MATCH;

BEGIN

!  Copy the Startup message into the CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;
```

```
!-----------------------------------------------------------
! Procedure to read the Startup message, and open the IN
! file, and open the $RECEIVE file.  This procedure also
! initializes the CPU^LIST array.
!-----------------------------------------------------------

PROC INIT;
BEGIN
   STRING .RECV^FILE[0:MAXFLEN - 1];
   INT     RECVLEN;
   STRING .TERMNAME[0:MAXFLEN - 1];
   INT     TERMLEN;
   INT     ERROR;
   INT     I;

!  Read Startup message and save it in global data area:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the IN file:

   ERROR := OLDFILENAME_TO_FILENAME_(
                       CI^STARTUP.INFILE.VOLUME,
                       TERMNAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_;

!  Open $RECEIVE:

   RECV^FILE ':=' "$RECEIVE" -> @S^PTR;
   RECVLEN := @S^PTR '-' @RECV^FILE;
   ERROR := FILE_OPEN_(RECV^FILE:RECVLEN,
                      RECVNUM,
                      !access!,
                      !exclusion!,
                      !nowait^depth!,
                      1);
   IF ERROR <> 0 THEN
      CALL FILE^ERRORS^NAME(RECV^FILE:RECVLEN,ERROR);


!  Initialize the CPU^LIST array:

   I := 2;
   DO
   BEGIN
      CPU^LIST[I] := 0;
      I := I + 1;
   END
   UNTIL I = TOP^CPU^NUMBER;

   CPU^LIST[0] := 1;
```

```
      CPU^LIST[1] := 1;
   END;
```

```
!------------------------------------------------------------
! Main procedure performs initialization, then goes into a
! loop in which it reads the $RECEIVE file and then calls the
! appropriate procedure depending on whether the message read
! was a system message, the message read was a user message,
! or the read generated an error.
!------------------------------------------------------------

PROC CMON^MAIN MAIN;
BEGIN
   INT BYTES^READ;
   INT ERROR;
   INT I;
   INT ERROR^CODE;

!  Initialize:

   CALL INIT;

!  Loop forever:

   WHILE 1 DO
   BEGIN
      ERROR := 0;

   !  Read a message from $RECEIVE and check for an error:

      CALL READUPDATEX(RECVNUM,SBUFFER,BUFSIZE,BYTES^READ);
      CALL FILE_GETINFO_(RECVNUM,ERROR);

      CASE ERROR OF
      BEGIN

      !  For a system message, reply with an error code
      !  of 0:

         6 -> BEGIN
            ERROR^CODE := 0;
            CALL REPLYX(!buffer!,
                        !write^count!,
                        !count^written!,
                        !message^tag!,
                        ERROR^CODE);
         END;
      !  For a user message, Select a procedure depending on
      !  the results of the read operation:

         0 -> BEGIN
            CASE BUFFER[0] OF
            BEGIN
               -60 -> CALL PROCESS^CONFIG^MSG;
               -59 -> CALL PROCESS^PRELOGON^MSG;
               -50 -> CALL PROCESS^LOGON^MSG;
               -51 -> CALL PROCESS^LOGOFF^MSG;
               -53 -> CALL PROCESS^ILLEGAL^LOGON^MSG;
               -57 -> CALL PROCESS^PASSWORD^MSG;
               -58 -> CALL PROCESS^REMOTEPASSWORD^MSG;
               -52 -> CALL PROCESS^PROCESSCREATION^MSG;
               -56 -> CALL PROCESS^ALTPRI^MSG;
               -54 -> CALL PROCESS^ADDUSER^MSG;
               -55 -> CALL PROCESS^DELUSER^MSG;
               50  -> CALL CHANGE^LOGON^MESSAGE;
               51  -> CALL CHANGE^LOGOFF^MESSAGE;
               61  -> CALL REJECT^REQUESTS;
               62  -> CALL ACCEPT^REQUESTS;
               63  -> CALL CHANGE^CPU^STATUS;
               OTHERWISE -> CALL UNEXPECTED^MESSAGE;
            END;
         END;

         OTHERWISE -> CALL FILE^ERRORS(RECVNUM);

      END;
   END;
END;
```

Sample Command-Interface Program

The command-interface program displays a menu allowing the operator to choose the run-time control function. A separate procedure processes each selection from the menu.

- The CONTROL^MAIN procedure calls INIT to perform initialization for the program and then enters a loop in which it displays a menu and calls a procedure to process the menu selection.
- The INIT and SAVE^STARTUP^MESSAGE procedures open the IN file and call CREATE^AND^OPEN^CMON to create the $CMON process if it does not exist and to open it. If $CMON does not exist, then this procedure creates it.
- The CREATE^AND^OPEN^CMON procedure calls OPEN^CMON to open the $CMON process. If $CMON does not exist, then this procedure creates it before calling OPEN^CMON. On return from OPEN^CMON, this procedure sends a Startup message to the new process, closes $CMON, then calls OPEN^CMON to open it again.

- The CHANGE^LOGON^MESSAGE procedure is called when the operator chooses to change the logon message by selecting 1 from the menu. This procedure prompts the operator for the new logon text before sending an RT^logon^message to $CMON.
- The CHANGE^LOGOFF^MESSAGE procedure is called when the operator chooses to change the logoff message by selecting 2 from the menu. This procedure prompts the operator for the new logoff text before sending an RT^logoff^message to $CMON.
- The REJECT^REQUESTS procedure is called when the operator chooses to disallow command-interpreter requests prior to shutting down the system. The operator selects 3 from the menu. The procedure prompts the operator for the new shutdown text and then sends an RT^shutdown^message to $CMON.
- The ACCEPT^REQUESTS procedure is called when the operator chooses to reenable command-interpreter requests by selecting 4 from the menu. The procedure sends an RT^start^message to $CMON.
- The CHANGE^CPU^STATUS procedure is called when the operator chooses to alter the status of a CPU by moving it from one priority group to another. The operator chooses to do this by selecting 5 from the menu. The procedure prompts the operator first for the CPU number, then for a 1 or a 0 depending on whether the operator wants to move the CPU into the high-priority group or the low-priority group.
- The EXIT^PROGRAM procedure is called when the operator chooses to quit the command-interface program by selecting x from the menu. This procedure stops the command-interface program.
- The ILLEGAL^REQUEST procedure is called whenever the operator makes an invalid selection from the menu. The procedure prints a message indicating that the selection is invalid before returning to the main procedure to redisplay the menu.
- The FILE^ERRORS and FILE^ERRORS^NAME procedures attempt to report any file-system errors that occur.
- The WRITE^LINE procedure provides a convenient way to display a line of text on the terminal.

The code for the command-interface program appears on the following pages.

```
?INSPECT, SYMBOLS, NOCODE, NOMAP
?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST

!Literals:

LITERAL TOP^CPU^NUMBER = 5;     !highest CPU number on system
LITERAL BUFSIZE = 512;          !size in bytes of I/O buffer
LITERAL MAXFLEN =
        ZSYS^VAL^LEN^FILENAME;  !maximum file-name length
LITERAL ABEND = 1;

!Global variables:

STRING .SBUFFER[0:BUFSIZE];     !I/O buffer
INT     TERMNUM;                !terminal file number
INT     CMONNUM;                !$CMON file number
STRING .S^PTR;                  !points to end of I/O buffer

STRUCT .CI^STARTUP;             !Startup message
BEGIN
   INT    MSGCODE;
   STRUCT DEFAULT;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
   END;
   STRUCT INFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILENAME[0:3];
   END;
   STRUCT OUTFILE;
   BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FILENAME[0:3];
   END;
   STRING PARAM[0:529];
END;
INT     MESSAGE^LEN;


?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,
?  FILE_OPEN_,WRITEX,FILE_GETINFO_,PROCESS_CREATE_,
?  PROCESS_GETPAIRINFO_,DNUMOUT,DNUMIN,WRITEREADX,
?  PROCESS_STOP_,FILE_CLOSE_,OLDFILENAME_TO_FILENAME_)
?LIST
```

```
!-------------------------------------------------------------
! The following DEFINEs help formatting and printing text.
!-------------------------------------------------------------

!  Initialize for a new line:

   DEFINE START^LINE = @S^PTR := @SBUFFER #;

!  Put a string into the line:

   DEFINE PUT^STR(S) = S^PTR ':=' S -> @S^PTR #;

!  Put an integer into the line:

   DEFINE PUT^INT(N) =
          @S^PTR := @S^PTR '+' DNUMOUT(S^PTR,$DBL(N),10) #;

!  Print the line:

   DEFINE PRINT^LINE =
          CALL WRITE^LINE(SBUFFER,@S^PTR '-' @SBUFFER) #;

!  Print a blank line:

   DEFINE PRINT^BLANK =
         CALL WRITE^LINE(SBUFFER,0) #;

!  Print a string:

   DEFINE PRINT^STR(S) = BEGIN START^LINE;
                               PUT^STR(S);
                               PRINT^LINE; END #;
```

```
!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameters are the file name, name length,
! and error number.  This procedure is mainly to be used when
! the file is not open, when there is no file number for it.
! File^ERRORS should be used when the file is open.
!
! The procedure also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS^NAME(FNAME:LEN,ERROR);
STRING   .FNAME;
INT       LEN;
INT       ERROR;
BEGIN

!  Compose and print the message:

   START^LINE;
   PUT^STR("File system error ");
   PUT^INT(ERROR);
   PUT^STR(" on file " & FNAME FOR LEN);

   CALL WRITEX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER);

!  Terminate the program:

   CALL PROCESS_STOP_(!process^handle!,
                      !specifier!,
                      ABEND);
END;


!------------------------------------------------------------
! Procedure for displaying file-system error numbers on the
! terminal.  The parameter is the file number.  The file name
! and error number are determined from the file number and
! FILE^ERRORS^NAME is then called to display the text.
!
! FILE^ERRORS^NAME also stops the program after displaying
! the error message.
!------------------------------------------------------------

PROC FILE^ERRORS(FNUM);
INT FNUM;
BEGIN
   INT      ERROR;
   INT     .FNAME[0:11];
   INT      LEN;

   CALL FILE_GETINFO_(FNUM,ERROR,FNAME:MAXFLEN,LEN);
   CALL FILE^ERRORS^NAME(FNAME:LEN,ERROR);
END;
```

```
!-------------------------------------------------------------
! Procedure to write a message on the terminal and check
! for any error.  If there is an error, the procedure tries
! to write a message about the error and the program is
! stopped.
!-------------------------------------------------------------

PROC WRITE^LINE(BUF,LEN);
STRING .BUF;
INT     LEN;
BEGIN
   CALL WRITEX(TERMNUM,BUF,LEN);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to generate an RT^logon^message.  This procedure
! prompts the operator for the logon display text, creates
! the RT^logon^message and sends it to the $CMON process.
!------------------------------------------------------------

PROC CHANGE^LOGON^MESSAGE;
BEGIN
   STRUCT .RT^LOGON^MESSAGE;      !structure to send to $CMON
   BEGIN
      INT MSGCODE;
      STRING DISPLAYTEXT[0:63];
   END;

   INT     BYTES^READ;

!  Set code for changing logon message in message data
!  structure:

   RT^LOGON^MESSAGE.MSGCODE := 50;

!  Blank the display text buffer:

   RT^LOGON^MESSAGE.DISPLAYTEXT[0] ':=' " ";
   RT^LOGON^MESSAGE.DISPLAYTEXT[1] ':='
      RT^LOGON^MESSAGE.DISPLAYTEXT[0] FOR 63;

!  Prompt operator for new logon text:

   SBUFFER ':=' "Enter logon message:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Put new logon text in message structure:

   RT^LOGON^MESSAGE.DISPLAYTEXT ':=' SBUFFER FOR BYTES^READ;

!  Send message to $CMON:

   CALL WRITEREADX(CMONNUM,RT^LOGON^MESSAGE,
                   $LEN(RT^LOGON^MESSAGE),2,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(CMONNUM);
END;
```

```
!--------------------------------------------------------------
! Procedure to generate a Change^logoff^message.  This
! procedure prompts the operator for the logoff display text,
! and then sends the new text to the $CMON procedure.
!--------------------------------------------------------------

PROC CHANGE^LOGOFF^MESSAGE;
BEGIN
   STRUCT .RT^LOGOFF^MESSAGE;      !data structure to send to
   BEGIN                           ! $CMON
      INT MSGCODE;
      STRING DISPLAYTEXT[0:63];
   END;

   INT     BYTES^READ;

!  Set message code in message structure for a change logoff
!  message:

   RT^LOGOFF^MESSAGE.MSGCODE := 51;

!  Blank the display text buffer:

   RT^LOGOFF^MESSAGE.DISPLAYTEXT[0] ':=' " ";
   RT^LOGOFF^MESSAGE.DISPLAYTEXT[1] ':='
      RT^LOGOFF^MESSAGE.DISPLAYTEXT[0] FOR 63;

!  Prompt the operator for the new logoff text:

   SBUFFER ':=' "Enter logoff message: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Put logoff text in message structure:

   RT^LOGOFF^MESSAGE.DISPLAYTEXT ':=' SBUFFER FOR BYTES^READ;

!  Send message to $CMON:

   CALL WRITEREADX(CMONNUM,RT^LOGOFF^MESSAGE,
                   $LEN(RT^LOGOFF^MESSAGE),2,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(CMONNUM);
END;
```

```
!-------------------------------------------------------------
! Procedure generates an RT^shutdown^message.  This procedure
! prompts the operator for the shutdown display text, puts
! it into the message, and sends the message to $CMON.
!-------------------------------------------------------------

PROC REJECT^REQUESTS;
BEGIN
   STRUCT .RT^SHUTDOWN^MESSAGE;        !structure to send to
   BEGIN                               ! $CMON
      INT MSGCODE;
      STRING SHUTDOWNTEXT[0:63];
   END;

   INT     BYTES^READ;

!  Set message code in message structure for shutdown
!  message:

   RT^SHUTDOWN^MESSAGE.MSGCODE := 61;

!  Blank the display text buffer:

   RT^SHUTDOWN^MESSAGE.SHUTDOWNTEXT[0] ':=' " ";
   RT^SHUTDOWN^MESSAGE.SHUTDOWNTEXT[1] ':='
      RT^SHUTDOWN^MESSAGE.SHUTDOWNTEXT[0] FOR 63;

!  Prompt the operator for the shutdown text:

   SBUFFER ':=' "Enter shutdown message: " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

!  Put the shutdown text into the message structure:

   RT^SHUTDOWN^MESSAGE.SHUTDOWNTEXT ':='
      SBUFFER FOR BYTES^READ;

!  Sends message to $CMON:

   CALL WRITEREADX(CMONNUM,RT^SHUTDOWN^MESSAGE,
                   $LEN(RT^SHUTDOWN^MESSAGE),2,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(CMONNUM);
END;
```

```
!------------------------------------------------------------
! Procedure to generate an RT^start^message and send it to
! the $CMON process.
!------------------------------------------------------------

PROC ACCEPT^REQUESTS;
BEGIN
   STRUCT RT^START^MESSAGE;       !structure to send to $CMON
   BEGIN
      INT MSGCODE;
   END;

   INT     BYTES^READ;

!  Set message code in message structure for the restart
!  message:

   RT^START^MESSAGE.MSGCODE := 62;

!  Send the structure to the $CMON process:

   CALL WRITEREADX(CMONNUM,RT^START^MESSAGE,
                   $LEN(RT^START^MESSAGE),
                   2,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(CMONNUM);
END;
```

```
!-------------------------------------------------------------
! Procedure to generate a CPU^changestatus^message.  This
! procedure prompts the operator for a processor number and then
! for a status (priority or nonpriority) to assign to that
! processor.  Finally, this procedure sends the
! CPU^changestatus^message to $CMON.
!-------------------------------------------------------------

PROC CHANGE^CPU^STATUS;
BEGIN
   STRING .EXT NEXT^ADR;
   INT     STATUS;
   INT     BYTES^READ;
   INT(32) NUMBER;

   STRUCT .CPU^CHANGESTATUS^MESSAGE;    !structure to send to
   BEGIN                                ! $CMON
      INT MSGCODE;
      INT PROCESSOR;
      INT STATUS;
   END;

!  Set the message code for changing processor status:

   CPU^CHANGESTATUS^MESSAGE.MSGCODE := 63;

!  Obtain a valid processor number from the operator:

   DO
   BEGIN
   PROMPT^AGAIN:
      SBUFFER ':=' "Enter valid processor number:  " -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,BYTES^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);
      SBUFFER[BYTES^READ] := 0;
      @NEXT^ADR := DNUMIN(SBUFFER,NUMBER,10,STATUS);
   END
   UNTIL STATUS = 0 AND $INT(NUMBER) >= 0
      AND $INT(NUMBER) <= TOP^CPU^NUMBER
      AND @NEXT^ADR = $XADR(SBUFFER[BYTES^READ]);

!  Set the processor number in the message structure:

   CPU^CHANGESTATUS^MESSAGE.PROCESSOR := $INT(NUMBER);
```

```
!  Obtain the new priority for the processor from the operator:

   DO
   BEGIN
      SBUFFER ':=' ["Enter new status: 1 for priority,",
                    "0 for non-priority: "]
                    -> @S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE,BYTES^READ);
      IF <> THEN CALL FILE^ERRORS(TERMNUM);

      SBUFFER[BYTES^READ] := 0;
      @NEXT^ADR := DNUMIN(SBUFFER,NUMBER,10,STATUS);
   END
   UNTIL STATUS = 0
         AND @NEXT^ADR = $XADR(SBUFFER[BYTES^READ])
         AND ($INT(NUMBER) = 1 OR $INT(NUMBER) = 0);

!  Put the new processor status into the message structure:

   CPU^CHANGESTATUS^MESSAGE.STATUS := $INT(NUMBER);

!  Send the message to $CMON:

   CALL WRITEREADX(CMONNUM,CPU^CHANGESTATUS^MESSAGE,
                   $LEN(CPU^CHANGESTATUS^MESSAGE),
                   2,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(CMONNUM);
END;


!------------------------------------------------------------
! Procedure to exit the process.
!------------------------------------------------------------

PROC EXIT^PROGRAM;
BEGIN
   CALL PROCESS_STOP_;
END;


!------------------------------------------------------------
! Procedure to respond to an invalid request.  This procedure
! displays a message and then returns to the main procedure
! to redisplay the menu.
!------------------------------------------------------------

PROC INVALID^REQUEST;
BEGIN
   PRINT^STR("Invalid request.  Please try again");
END;
```

```
!------------------------------------------------------------
! Procedure to prompt the user for the next function to
! perform:
!
! "1" to change the logon message
! "2" to change the logoff message
! "3" to prohibit requests during shutdown
! "4" to re-enable requests
! "5" to change the processor status
! "x" to exit the program
!
! The selection made is returned as the result of the call.
!-------------------------------------------------------------

INT PROC GET^COMMAND;
BEGIN
   INT  BYTES^READ;

!  Prompt the user for the function to be performed:

   PRINT^BLANK;
   PRINT^STR("Type '1' to change the logon message ");
   PRINT^STR("     '2' to change the logoff message ");
   PRINT^STR("     '3' to prohibit requests in shutdown ");
   PRINT^STR("     '4' to re-enable requests ");
   PRINT^STR("     '5' to change the processor status ");
   PRINT^STR("     'x' to exit the program ");

   SBUFFER ':=' "Choice:  " -> @S^PTR;
   CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                   BUFSIZE,BYTES^READ);
   IF <> THEN CALL FILE^ERRORS(TERMNUM);

   SBUFFER[BYTES^READ] := 0;
   RETURN SBUFFER[0];
END;
```

```
!------------------------------------------------------------
! Procedure opens the $CMON process.  Prompts the user to try
! again if the open fails.
!------------------------------------------------------------

PROC OPEN^CMON(PROCESS^NAME,PROCESS^NAMELEN,SERVER^NUM);
STRING .PROCESS^NAME;
INT     PROCESS^NAMELEN;
INT    .SERVER^NUM;

BEGIN
   INT  ERROR;

TRY^AGAIN:
   ERROR := FILE_OPEN_(PROCESS^NAME:PROCESS^NAMELEN,
                       SERVER^NUM);
   IF ERROR <> 0 THEN
   BEGIN
      PRINT^STR("Could not open $CMON");
      SBUFFER ':=' "Do you wish to try again? (y/n): "
                  ->@S^PTR;
      CALL WRITEREADX(TERMNUM,SBUFFER,@S^PTR '-' @SBUFFER,
                      BUFSIZE);
      IF (SBUFFER[0] = "n") OR (SBUFFER[0] = "N") THEN
         CALL PROCESS_STOP_
         ELSE GOTO TRY^AGAIN;
   END;
END;
```

```
!------------------------------------------------------------
! Procedure handles creating and opening a $CMON process.  If
! $CMON already exists it calls OPEN^CMON to open it.  If
! $CMON does not exist, it creates the process and sends it
! the standard process initialization sequence.
!------------------------------------------------------------

PROC CREATE^AND^OPEN^CMON(CMON^NUM,CMON^OBJECT^NAME,
                          OBJFILE^NAMELEN,PROCESS^NAME,
                          PROCESS^NAMELEN);
INT    .CMON^NUM;                !file number of $CMON process
STRING .CMON^OBJECT^NAME;        !name of $CMON object file
INT     OBJFILE^NAMELEN;         !file-name length
STRING .PROCESS^NAME;            !name of $CMON process
INT     PROCESS^NAMELEN;         !process-name length

BEGIN
   INT    ERROR;

!  Check whether $CMON already running.  If so, open it.
!  If not, create it and open it:

   ERROR := PROCESS_GETPAIRINFO_(
                    ,PROCESS^NAME:PROCESS^NAMELEN);

!  IF $CMON already exist, open it:

   CASE ERROR OF
   BEGIN

      0, 4 -> BEGIN

      !  The process already exists; open it:

         CALL OPEN^CMON(PROCESS^NAME,PROCESS^NAMELEN,
                        CMON^NUM);

      END;

      9 -> BEGIN

      !  The process does not exist, create it and open it,
      !  send it a Startup message, close it, and then reopen
      !  it:
```

```
     !  Create process:

        ERROR := PROCESS_CREATE_(
                         CMON^OBJECT^NAME:OBJFILE^NAMELEN,
                         !library^file:length!,
                         !swap^file:length!,
                         !ext^swap^file:length!,
                         !priority!,
                         !processor!,
                         !process^handle!,
                         !error^detail!,
                         ZSYS^VAL^PCREATOPT^NAMEINCALL,
                         PROCESS^NAME:PROCESS^NAMELEN);
        IF ERROR <> 0 THEN
        BEGIN
           PRINT^STR("Unable to create $CMON");
           CALL PROCESS_STOP_;
        END;

     !  Open the new $CMON process:

        CALL OPEN^CMON(PROCESS^NAME,PROCESS^NAMELEN,
                       CMON^NUM);

     !  Send $CMON a Startup message:

        CI^STARTUP.MSGCODE := -1;
        CALL WRITEX(CMON^NUM,CI^STARTUP,MESSAGE^LEN);
        IF <> THEN
        BEGIN
           CALL FILE_GETINFO_(CMON^NUM,ERROR);
           IF ERROR <> 70 THEN
              PRINT^STR("Could not write Start-Up " &
                         "message to server ");
         END;

      !  Close $CMON:

         ERROR := FILE_CLOSE_(CMON^NUM);

      !  Reopen $CMON:

         CALL OPEN^CMON(PROCESS^NAME,PROCESS^NAMELEN,
                        CMON^NUM);
      END;

      OTHERWISE -> BEGIN

      !  Unexpected error return from PROCESS_GETPAIRINFO_:

         PRINT^STR("Unexpected error ");
      END;
   END;
END;
```

```
!-------------------------------------------------------------
! Procedure to save the Startup message in the global
! data area.
!-------------------------------------------------------------

PROC SAVE^STARTUP^MESSAGE(RUCB,START^DATA,MESSAGE,
                          LENGTH,MATCH)VARIABLE;
INT  .RUCB;
INT  .START^DATA;
INT  .MESSAGE;
INT   LENGTH;
INT   MATCH;

BEGIN

!  Save Startup message in CI^STARTUP structure:

   CI^STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
   MESSAGE^LEN := LENGTH;
END;
```

```
!-------------------------------------------------------------
! Procedure to read the Startup message and open the terminal
! and $CMON files.
!-------------------------------------------------------------

PROC INIT;
BEGIN
   STRING  .PROGNAME[0:MAXFLEN - 1];
   INT      PROGNAME^LEN;
   STRING  .PROCESS^NAME[0:MAXFLEN - 1];
   INT      PROCESS^NAME^LEN;
   STRING  .TERMNAME[0:MAXFLEN - 1];
   INT      TERMLEN;
   INT      ERROR;

!  Call the INITIALIZER and save the Startup message:

   CALL INITIALIZER(!rucb!,
                    !passthru!,
                    SAVE^STARTUP^MESSAGE);

!  Open the IN file from the Startup message:

   ERROR := OLDFILENAME_TO_FILENAME_(
                           CI^STARTUP.INFILE.VOLUME,
                           TERMNAME:MAXFLEN,TERMLEN);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);
   ERROR := FILE_OPEN_(TERMNAME:TERMLEN,TERMNUM);
   IF ERROR <> 0 THEN CALL PROCESS_STOP_(!process^handle!,
                                         !specifier!,
                                         ABEND);

!  Open the $CMON; create it if it does not already exist:

   PROGNAME ':=' "$APPLS.PROGS.ZCMON" ->@S^PTR;
   PROGNAME^LEN := @S^PTR '-' @PROGNAME;
   PROCESS^NAME ':=' "$CMON" -> @S^PTR;
   PROCESS^NAME^LEN := @S^PTR '-' @PROCESS^NAME;
   CALL CREATE^AND^OPEN^CMON(CMONNUM,PROGNAME,PROGNAME^LEN,
                             PROCESS^NAME,PROCESS^NAME^LEN);
END;
```

```
!-------------------------------------------------------------
! Main procedure performs initialization, then goes into a
! loop in which it reads the $RECEIVE file and then calls the
! appropriate procedure depending on whether the message read
! was a system message, the message used was a user message,
! or the read generated an error.
!-------------------------------------------------------------

PROC CONTROL^MAIN MAIN;
BEGIN
   INT    I;
   STRING CMD;

!  Open terminal and $CMON:

   CALL INIT;

!  Loop forever:

   WHILE 1 DO
   BEGIN

   !  Prompt user for function to perform:

      CMD := GET^COMMAND;


   !  Select a procedure depending on value returned from
   !  GET^COMMAND:

      CASE CMD OF
      BEGIN

         "1"  -> CALL CHANGE^LOGON^MESSAGE;

         "2"  -> CALL CHANGE^LOGOFF^MESSAGE;

         "3"  -> CALL REJECT^REQUESTS;

         "4"  -> CALL ACCEPT^REQUESTS;

         "5"  -> CALL CHANGE^CPU^STATUS;

         "x"  -> CALL EXIT^PROGRAM;

         OTHERWISE -> CALL INVALID^REQUEST;
      END;
   END;
END;
```

Debugging a TACL Monitor ($CMON)

Replacing a standard $CMON with an untested program being debugged can lead to unacceptable delays and inconvenience to the user community. You should therefore name the TACL monitor program you are developing something other than $CMON and follow the guidelines described in this subsection for debugging and testing.

Debugging and testing is made more difficult because TACL processes always make requests to a process named $CMON—the TACL program is hard coded to do this. Therefore you should use a TACL macro or some other program that simulates the TACL part of the TACL/$CMON interface. A TACL Macro for Debugging and Testing a $CMON Program, following, provides an example.

A TACL Macro for Debugging and Testing a $CMON Program

The following example shows a TACL macro that you can use for sending TACL messages to a TACL monitor program and receiving and displaying the reply. This example tests the TACL monitor for its response to Prelogon^msg, Config^msg, and Logon^msg messages. It can easily be modified to test the response to other TACL messages.

```
?TACL MACRO
==
==  This TACL macro can be used as a template to test $CMON
==  processes
==

#FRAME

==  Structure for sending TACL messages to $CMON:

[#DEF to^cmon STRUCT
  BEGIN
  INT   message^code VALUE     0;
  BYTE  group        VALUE   101;
  BYTE  user         VALUE   131;
  INT   cipri        VALUE     0;
  FNAME ciinffile    VALUE $ztnt.#pty21;
  FNAME cioutfile    VALUE $ztnt.#pty21;
  END;
]

==  Structure for receiving the reply to a Prelogon^msg
==  message:

[#DEF prelogon^reply STRUCT
  BEGIN
  INT   reply^code;
  CHAR  reply^text (0:131);
  END;
]
```

```
==  Structure for receiving the reply to a Logon^msg message:

[#DEF loggedon^reply STRUCT
  BEGIN
  INT   reply^code;
  CHAR  reply^text (0:131);
  END;
]

==  Structure for receiving a reply to a Config^msg message.
==  cmon:configreply^text is used when $CMON accepts the
==  default parameters.  cmon:tacl^config is used when $CMON
==  changes the configuration parameters:

[#DEF cmon STRUCT
  BEGIN
  STRUCT configreply^text;
    BEGIN
    INT   reply^code;
    CHAR  reply^text (0:131);
    END;
  STRUCT tacl^config REDEFINES configreply^text;
    BEGIN
    INT reply^code;
    INT count;
    INT autologoffdelay;
    INT logoffscreenclear;
    INT remotesuperid;
    INT blindlogon;
    INT namelogon;
    INT cmontimeout;
    INT cmonrequired;
    INT remotecmontimeout;
    INT remotecmonrequired;
    INT nochangeuser;
    INT stoponfemodemerr;
    INT requestcmonuserconfig;
    END;
  END;
]

[#PUSH status r^error r^rec prompt
  cmon^process^name
  cmon^process^sock
]
```

```
==  Open $CMON socket:

#SET cmon^process^name $CMOO      == Temporary name of
                                  == $CMON process
#SET cmon^process^sock            == .#cmon
#SET status [#REQUESTER/WAIT 5000/READ &
    [cmon^process^name][cmon^process^sock] &
    r^error r^rec prompt]

==  Send a Prelogon^msg request and display the reply:

#SET to^cmon:message^code -59
#APPENDV prompt to^cmon
#EXTRACTV r^rec prelogon^reply
#OUTPUTV prelogon^reply

==  Send a Config^msg request and display the reply:

#SET to^cmon:message^code -60
#APPENDV prompt to^cmon
#EXTRACTV r^rec cmon
[#IF [cmon:tacl^config:reply^code] |THEN|
  #OUTPUTV  cmon:configreply^text
|ELSE|
  #OUTPUTV cmon:tacl^config
]

==  Send a Logon^msg request and display the reply:

#SET to^cmon:message^code -50
#APPENDV prompt to^cmon
#EXTRACTV r^rec loggedon^reply
#OUTPUTV loggedon^reply

==  Close $CMON:

#SET status [#REQUESTER/WAIT 5000/CLOSE r^rec]
#UNFRAME
```

For details of TACL programming techniques, see the *TACL Programming Guide*.

Procedure for Debugging and Testing a TACL Monitor ($CMON)

The following steps describe the recommended way to debug or test a $CMON program.

1. Start your $CMON program and give it a name other than $CMON. Start it under debug if desired:

   ```
   1> RUN[D] ZCMON/NAME $CMOO/
   ```

   See the *Inspect Manual* or the *Debug Manual* for general debugging information.

2. Start your TACL macro:

   ```
   1> run cmootest
   PRELOGON^REPLY(0)
     REPLY^CODE(0:0) 0
     REPLY^TEXT(0:131)

   CONFIGREPLY^TEXT(0)
     REPLY^CODE(0:0) 1
     REPLY^TEXT(0:131)

   LOGGEDON^REPLY(0)
     REPLY^CODE(0:0) 0
     REPLY^TEXT(0:131)
                     Logon accepted
   ```

   The display shows the contents of the Prelogon^reply, Config^text^reply, and Logon^reply messages as returned by the new $CMON process.

Note that when you run your command interface program, CMONCOM, you must also change its name for the $CMON program.

24 Writing a Terminal Simulator

Using the interprocess-communication features described in Section 6, Communicating With Processes, you can write a program that simulates a terminal. A user process can communicate with this terminal-simulation process as though it were a real terminal. This user process is typically a requester in an application designed according to the requester/server model.

Generally, a terminal-simulation process provides an interface between a requester process and one or more real terminals. Some of the functions a terminal-simulation process could include are:

- Multiplexing: mapping input/output requests from one or more requesters to one or more terminals
- Translating: changing certain characters during the data transfer between a requester and a real terminal
- Filtering: adding, removing, or altering information during the data transfer between a requester and a real terminal

When writing a terminal-simulation program, you must give the terminal-simulation process the following properties:

- The terminal-simulation process must have a device subtype of 30.
- The terminal-simulation process must be named.
- The terminal-simulation process must accept system messages through its $RECEIVE file.
- The terminal-simulation process must specify how Setmode and Setparam messages are to be handled.
- The terminal-simulation process must handle WRITE[X], READ[X], and WRITEREAD[X] I/O requests from requesters.
- The terminal-simulation process must handle system messages such as:
    - Device type information request messages
    - Setmode messages
    - Setparam messages
    - Control messages
- The terminal-simulation process must handle the BREAK key.

The remainder of this section discusses the guidelines above in detail.

Specifying Device Subtype 30

A terminal-simulation process must have a device subtype of 30. The device subtype of a process is an attribute that is stored in the object file. By default, the subtype is 0; however, you can specify some value for this attribute when you compile or bind the program.

Why Device Subtype 30 Must Be Specified

Specifying device subtype 30 tells the system that the terminal-simulation process will supply device information in response to a request for the device-type information.

A requester that communicates with a real terminal can call a procedure such as FILE_GETINFO_ to get the device type of that device. For a real terminal, the FILE_GETINFO_ procedure returns a device type of 6 to the requester. Processes normally return device type 0; however, a terminal simulator must be able to return device type 6 just like a real terminal. This operation is called device-type substitution.

To allow the terminal-simulation process to specify its own device type, you must specify device subtype 30 for the process. Specifying device subtype 30 for the terminal-simulation process specifies that the process performs device-type substitution.

The mechanism for your terminal-simulation program to return the device type itself is described later in this section under Processing System Messages.

How to Specify Device Subtype 30

You can specify the subtype attribute using either a compiler directive or a linker command:

- You can place a compiler directive at the beginning of your TAL program:

  ```
  ?SUBTYPE 30
  ```

- For a TNS or accelerated program, you can use the SET SUBTYPE command in Binder when linking the program:

  ```
  @ SET SUBTYPE 30
  ```

- For a TNS/R native program, you can use the set subtype command in the nld utility when linking the program. The following nld command links two object files, ofile1 and ofile2, sets the subtype attribute of the resultant object file to 30, and names it objfile:

  ```
  30> nld ofile1 ofile2 -set subtype 30 -o objfile
  ```

Assigning a Name to the Terminal-Simulation Process

A terminal-simulation process must be a named process. You assign a name to the process when you call the PROCESS_CREATE_ procedure to create the process or type the TACL RUN command. The following example creates a process named "$T1."

```
OBJFILE ':=' "TERMFILE" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR ‘-’ @OBJFILE;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEINCALL;
PROCESS^NAME ':=' "$T1";
PROCESS^NAME^LENGTH := 3;
ERROR := PROCESS_CREATE_(OBJFILE:OBJFILENAME^LENGTH,
                         !library^file:length!,
                         !swap^file:length!,
                         !ext^swap^file:length!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         PROCESS^NAME:PROCESS^NAME^LENGTH,
                         PROCESS^DESCRIPTOR:MAXLEN,
                         PROCESS^DESCRIPTOR^LENGTH);
```

See Section 16, Creating and Managing Processes, for more details on creating named processes.

Accepting System Messages Through $RECEIVE

A terminal-simulation process must accept system messages in its $RECEIVE file. The process might receive a variety of system messages. Among these are:

- Requests for device-type information
- Setmode messages
- Setparam messages
- Control messages

To receive and reply to these messages, you must specify that the terminal-simulation process accepts system messages.

The system messages listed above are sent to $RECEIVE when a FILE_OPEN_ call is executed. The system messages cannot be blocked. By default, bit 15 is set to the default value 0 of the options parameter in the FILE_OPEN_ call that opens $RECEIVE:

```
FILE^NAME ':=' "$RECEIVE" -> @S^PTR;
LENGTH := @S^PTR '-' @FILE^NAME;
OPTIONS.<15> := 0;
ERROR := FILE_OPEN_(FILE^NAME:LENGTH,
                    RECV^NUM);
```

See Section 6, Communicating With Processes, for more information about opening $RECEIVE.

Specifying How to Process System Messages

A terminal-simulation process can specify how certain system messages are to be handled. To do this, the process must call SETMODE function 80.

Using SETMODE function 80, the terminal-simulation process can do the following:

- Specify whether a requester can include the *last-params* parameter in SETMODE procedure calls
- Specify whether a requester can call the SETPARAM procedure
- Specify whether the terminal-simulation process accepts cancellation messages

The following paragraphs discuss allowing the *last-params* parameter in SETMODE procedure calls and allowing SETPARAM calls. For a discussion of accepting cancellation messages, see Section 6, Communicating With Processes.

Allowing the Requester to Specify the *last-params* Parameter

A terminal-simulation process can call SETMODE function 80 to specify whether a requester can specify the *last-params* parameter in SETMODE procedure calls made against the terminal-simulation process.

When a requester communicates with a real terminal, the requester can call the SETMODE procedure to control the operation of the real terminal. The requester can specify the *last-params* parameter when calling SETMODE. The system uses *last-params* to return to the requester the parameter values specified with the last SETMODE call that specified the same function as the current SETMODE call.

When a requester communicates with a terminal-simulation process, its SETMODE calls cause Setmode messages to be sent to the terminal-simulation process. The process must read, process, and reply to these messages.

When a requester communicates with a terminal-simulation process, by default it cannot include the *last-params* parameter in its SETMODE calls. However, the terminal-simulation process can call SETMODE function 80 to allow requesters to include the *last-params* parameter in SETMODE calls.

If the terminal-simulation process allows requesters to specify the *`last-params`* parameter, the process must remember the previous parameter values for SETMODE functions. Furthermore, the process must return this information to the requester when responding to Setmode messages.

Reading and responding to Setmode messages is discussed in detail later in this section.

To allow a requester to include the *`last-params`* parameter on SETMODE calls, assign 1 to bit 15 of *`param1`* in a SETMODE function 80 call:

```
PARAM1.<15> := 1;
CALL SETMODE(REQ^NUM,80,PARAM1);
```

Allowing the Requester to Call SETPARAM

A terminal-simulation process can call SETMODE function 80 to specify whether the requester can call the SETPARAM procedure against the terminal-simulation process.

A requester that communicates with a real terminal can call SETPARAM to control the operation of the terminal. Most SETPARAM functions are related to data communication; however, SETPARAM function 3 specifies the owner of the BREAK key. (SETPARAM function 3 works like SETMODE function 11, except that SETPARAM function 3 can also specify a BREAK tag.) When a requester communicates with a terminal-simulation process, by default it cannot call the SETPARAM procedure. However, the terminal-simulation process can call SETMODE function 80 to allow requesters to call the SETPARAM procedure.

If a terminal-simulation process allows requesters to call SETPARAM, a Setparam message is sent to the process when the requester calls SETPARAM. The process must read and respond to Setparam messages. The response to the Setparam message must include the previous parameter values for the particular SETPARAM function.

Reading and responding to Setparam messages is discussed in detail later in this section.

To allow a requester to call SETPARAM, assign 1 to bit 14 of the *`param1`* parameter in a SETMODE function 80 call:

```
PARAM1.<14> := 1;
CALL SETMODE(REQ^NUM,80,PARAM1);
```

Processing I/O Requests

A terminal-simulation process accepts and responds to I/O requests from one or more requesters. I/O requests are received through the $RECEIVE file.

Immediately after receiving a message from $RECEIVE, the process should check the condition code. If a CCG condition code is returned, then the process should call the FILE_GETINFO_ procedure to retrieve the file-system error number. File-system error

6 indicates that a file-system message was received, so the message should be processed as a system message.

The following code fragment checks the message read from $RECEIVE to determine whether it is a system message or a user message:

```
CALL READUPDATE(RECV^NUM,BUFFER,RCOUNT);
CALL FILE_GETINFO_(RECV^NUM,ERROR);
IF ERROR = 6 THEN...                  !system message
IF ERROR = 0 THEN...                  !user message
```

If the condition code is normal, the process must determine whether the received message is an I/O request and, if so, the type of I/O request. You can accomplish this by calling the FILE_GETRECEIVEINFO_ procedure.

The FILE_GETRECEIVEINFO_ procedure returns information about the last message read from $RECEIVE. The following values returned by the FILE_GETRECEIVEINFO_ procedure are particularly important to a terminal-simulation process:

- The process handle of the process that sent the message.

  The terminal-simulation process might use the process handle to communicate with the requester process. For example, the terminal-simulation process might need to send a BREAK message to the requester process. The process handle of the requester is required in the BREAKMESSAGE_SEND_ procedure call.

- A message tag associated with this message.

  If the terminal-simulation process reads several messages before issuing a reply, then the simulation process can use the message tag to specify the message to which it is replying.

- The maximum number of bytes the requester expects to read.

  The terminal-simulation process can use this value to determine the maximum number of bytes to return to the requester in response to a READ[X] or WRITEREAD[X] request.

- A value indicating the type of I/O request received.

  The terminal-simulation process can use this value to determine whether the most recently received message is an I/O request and, if so, whether it is a READ[X] request, a WRITE[X] request, or a WRITEREAD[X] request.

The FILE_GETRECEIVEINFO_ procedure returns other parameters in addition to these; see the *Guardian Procedure Calls Reference Manual* for a complete description of the FILE_GETRECEIVEINFO_ procedure.

Immediately after reading a message from $RECEIVE, the terminal-simulation process should call FILE_GETRECEIVEINFO_ to check the value returned for the I/O type. The meanings of the I/O type values that can be returned are as follows:

0 The most recent message read from $RECEIVE was not an I/O request.

The terminal-simulation process should process the message as a system message.

1 The most recent message read from $RECEIVE was a WRITE[X] request.

The buffer for the read from $RECEIVE contains the data that was written by the requester. The *count-read* parameter for the read from $RECEIVE indicates the number of bytes written by the requester.

2 The most recent message read from $RECEIVE was a READ[X] request.

The buffer for the read from $RECEIVE contains no data. The maximum reply count returned by the FILE_GETRECEIVEINFO_ procedure indicates the maximum number of bytes the requester expects to read.

3 The most recent message read from $RECEIVE was a WRITEREAD[X] request.

The buffer for the read from $RECEIVE contains the data that was written by the requester, if any. The *count-read* parameter for the read from $RECEIVE indicates how many bytes were written by the requester. The maximum reply count returned by the FILE_GETRECEIVEINFO_ procedure indicates the maximum number of bytes the requester expects to read.

After the terminal-simulation process performs any processing associated with the request, it must respond to the requester. Responding to the requester is performed by calling the REPLY[X] procedure.

You can use the *error-return* parameter of the REPLY[X] procedure to return error codes to the requester. The error number you specify in the *error-return* parameter also controls the condition code returned to the requester.

The following example checks the I/O type of a request received from the requester. If it is a WRITE[X] request, the code sends the message to the terminal. If the request is for a READ operation, then the code fragment reads information from the terminal and returns it to the requester. For a WRITEREAD[X] request, the code fragment writes the message to the terminal, waits for a response, and then writes the response back to the requester process.

```
  .
  .
CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
CALL FILE_GETINFO_(RECV^NUM,ERROR);
CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);
IF ERROR = 6 THEN                     !system message
BEGIN
  .
  .
END;
IF ERROR = 0 THEN                     !user message
BEGIN
   PROCESS^HANDLE ':=' RECEIVE^INFO[6] FOR 10;
   MESSAGE^TAG := RECEIVE^INFO[2];
   BYTES^TO^RETURN := RECEIVE^INFO[1];
   IO^TYPE := RECEIVE^INFO[0];
   CASE OF IO^TYPE
   BEGIN

   !  I/O is a WRITE; send message on to the terminal and
   !  reply to requester:

      1 -> BEGIN
         ERROR^RETURN := 0;
         WCOUNT := BYTES^READ;
         CALL WRITEX(TERM^NUM,SBUFFER,WCOUNT);
         IF <> THEN
            CALL FILE_GETINFO_(TERM^NUM,ERROR^RETURN);
         CALL REPLYX(!buffer!,
                     !write^count!,
                     !count^written!,
                     !message^tag!,
                     ERROR^RETURN);
      END;

   !  I/O is a READ; read message from terminal and reply to
   !  requester process:

      2 -> BEGIN
         ERROR^RETURN := 0;
         RCOUNT := BYTES^TO^RETURN;
         CALL READX(TERM^NUM,SBUFFER,RCOUNT,BYTES^READ);
         IF <> THEN
            CALL FILE_GETINFO_(TERM^NUM,ERROR^RETURN);
         CALL REPLYX(SBUFFER,
                     BYTES^READ,
                     !count^written!,
                     !message^tag!,
                     ERROR^RETURN);
      END;
   !  I/O is a WRITEREAD[X]; write message to terminal, wait
   !  for response, write response back to requester:

      3 -> BEGIN
         ERROR^RETURN := 0;
         WCOUNT := BYTES^READ;
         RCOUNT := BYTES^TO^RETURN;
         CALL WRITEREADX(TERM^NUM,SBUFFER,WCOUNT,RCOUNT,
                         BYTES^READ);
         IF <> THEN
            CALL FILE_GETINFO_(TERM^NUM,ERROR^RETURN);
         CALL REPLYX(SBUFFER,
                     BYTES^READ,
                     !count^written!,
                     !message^tag!,
                     ERROR^RETURN);
      END;

      !Any other I/O type is unexpected:
      OTHERWISE -> BEGIN
       .
       .
      END;
   END;
END;
```

For more information about the REPLY[X] procedure, see Section 6 of this manual Communicating With Processes, or the *Guardian Procedure Calls Reference Manual*.

Processing System Messages

A terminal-simulation process might receive system messages through its $RECEIVE file. In particular, the process might receive these system messages:

-32	Control message
-33	Setmode message
-37	Setparam message
-106	Request for device-type information (D-series format message)
-147	Request for device configuration information (G-series format message)

The following paragraphs describe when these messages are sent to the terminal-simulation process and discuss how the terminal-simulation process should respond (except for system messages -147). For description of system message -147, refer to the *Guardian Procedure Errors and Messages Manual*.

Processing Control Messages

When the requester calls the Control procedure, a Control message is sent to the terminal-simulation process. The message has the following format:

Structure of the Control message (system message -32):

```
sysmsg[0] = -32
sysmsg[1] = the operation parameter of the CONTROL call
sysmsg[2] = the param parameter of the CONTROL call
```

A requester can perform many terminal-related operations by calling the CONTROL procedure. Your terminal-simulation process must determine the appropriate processing to perform for each CONTROL operation.

Processing Setmode Messages

When the requester process calls the SETMODE procedure, a Setmode system message is sent to the terminal-simulation process. This message has the following format:

Structure of a Setmode system message (message -33):

```
sysmsg[0]       = -33
sysmsg[1]       = SETMODE function code
sysmsg[2]       = param1 of the SETMODE call
sysmsg[3]       = param2 of the SETMODE call
sysmsg[4].<13> = set to 1 if param1 was specified
sysmsg[4].<14> = set to 1 if param2 was specified
sysmsg[4].<15> = set to 1 if last-params was specified
```

However, if the terminal-simulation process did not call SETMODE function 80 to allow requesters to include the *last-params* parameter on SETMODE calls, then system message -33 does not include sysmsg[4].

If the terminal-simulation process called SETMODE function 80 to allow the *last-params* parameter on SETMODE calls, then the process can use sysmsg[4] to determine the parameters the requester included in the last SETMODE call that used the same function value. For example, if sysmsg[4].<15> is set to 1, then the terminal-simulation process must determine the previous values for this SETMODE function and return those values in its response. (The value representing the previous *param1* can also be an internally defined value that identifies the previous owner of BREAK; refer to Tracking the BREAK Owner, later in this section.) If this bit is set to 0, the terminal-simulation process need not return previous parameter values for this

SETMODE function. The terminal-simulation process must respond to system message -33 with a message in the following format:

Structure of -33 reply message:

```
replymsg[0] = -33
replymsg[1] = previous value of param1, if requested
replymsg[2] = previous value of param2, if requested
```

A requester can perform many terminal-related functions by calling the SETMODE procedure. Your terminal-simulation process must determine the appropriate processing to perform for each SETMODE function.

Processing Setparam Messages

If the terminal-simulation process used SETMODE function 80 to allow requesters to call SETPARAM, the process must read and respond to Setparam messages.

A Setparam message has the following format:

Structure of a Setparam message (system message -37):

```
sysmsg[0]       = -37
sysmsg[1]       = SETPARAM function code
sysmsg[2].<14> = set to 1 if param-array was specified in
                  SETPARAM call
sysmsg[2].<15> = set to 1 if last-param-array was specified
                  in the SETPARAM call
sysmsg[3]       = the param-count parameter of the SETPARAM
                  call
sysmsg[4:n]     = the param-array parameter of the SETPARAM
                  call
```

The terminal-simulation process must respond to system message -37 with a message in the following format:

Structure of -37 reply message:

```
replymsg[0] = -37
replymsg[1] = value for last-param-count
replymsg[2] = value for last-param-array
```

A requester can perform one terminal-related function by calling the SETPARAM procedure. That function is SETPARAM function 3, which specifies the owner of the BREAK key. Your terminal-simulation process should handle SETPARAM function 3 in a manner similar to the way it handles SETMODE function 11; however, in processing SETPARAM function 3, you must also process the BREAK tag information.

The SETPARAM functions are described in the *Guardian Procedure Calls Reference Manual*.

Processing Device-Type Information Requests

A requester process can call a procedure such as FILE_GETINFO[BYNAME]_ to determine the device type of the terminal-simulation process. When another process requests device-type information from the terminal-simulation process, system message -106 is sent to the terminal-simulation process. The format of system message -106 is as follows:

Structure of system message -106:

```
sysmsg[0]               = -106
sysmsg[1:3]             = reserved
sysmsg[4]               = length in bytes of the qualifier
                          name
sysmsg[5] FOR sysmsg[4] = the qualifier name
```

The terminal-simulation process must reply with a message in the following format:

Structure of -106 reply message:

```
replymsg[0]         = -106
replymsg[1]         = device type
replymsg[2]         = device subtype
replymsg[3:5]       = reserved (filled with -1)
replymsg[6]         = physical-record length
```

The terminal-simulation process should return 6 in replymsg[1] to indicate the device type for a terminal and 30 in replymsg[2] to indicate device substitution. That is, the device-type value for a terminal is substituted for the device-type value for a process.

The following code fragment processes requests for device-type information:

```
STRUCT REPLY^TO^106;
BEGIN
   INT MESSAGE^NUMBER;
   INT DEVICE^TYPE;
   INT DEVICE^SUBTYPE;
   INT RESERVED[0:2];
   INT RECORD^LEN;
END;
 .
 .


CALL READUPDATEX(RECV^NUM,SBUFFER,RCOUNT,BYTES^READ);
CALL FILE_GETINFO_(RECV^NUM,ERROR);
CALL FILE_GETRECEIVEINFO_(RECEIVE^INFO);
IF ERROR = 6 THEN                     !system message
BEGIN

   IF SBUFFER[0] = -106 THEN

   !It is a request for device-type information:
```

```
BEGIN

   !Set device type to 6 and subtype to 30:
   REPLY^TO^106.MESSAGE^NUMBER := -106;
   REPLY^TO^106.DEVICE^TYPE := 6;
   REPLY^TO^106.DEVICE^SUBTYPE := 30;
   REPLY^TO^106.RESERVED[0] := -1;
   REPLY^TO^106.RESERVED[1] := -1;
   REPLY^TO^106.RESERVED[2] := -1;
   REPLY^TO^106.RECORD^LEN := <configured record length for
                               terminal>

   !Respond to the requester:
   CALL REPLYX(REPLY^TO^106,$LEN(REPLY^TO^106));
END;
```

Managing the BREAK Key

A real terminal has a BREAK key that, when pressed, sends a Break-on-device message to the requester that communicates with the terminal. The requester can control BREAK handling as follows:

- The requester can call SETMODE function 11 or SETPARAM function 3 to specify the owner of the BREAK key. The owner of the BREAK key is the process that receives the Break-on-device message when the terminal user presses the BREAK key. The BREAK key can be owned by only one process.
- The process that owns the BREAK key can call SETMODE function 12 to place a terminal in BREAK mode or normal mode. When the terminal is in BREAK mode, only a process that has BREAK access can communicate with the terminal. When a terminal is in normal mode, any process that has the terminal open can communicate with the terminal.

Typically, the BREAK key owner places the terminal in BREAK mode when it receives a Break-on-device message.

When writing a terminal-simulation process, you must include the following logic to manage the BREAK key:

- The terminal-simulation process must know the current BREAK key owner.
- The terminal-simulation process must communicate with other processes depending on whether the terminal-simulation process is in BREAK mode or normal mode.
- The terminal-simulation process can send Break-on-device messages to requester processes.

The following paragraphs discuss these topics.

Tracking the BREAK Owner

The terminal-simulation process must know the BREAK owner so that it can send Break-on-device messages to it.

When the terminal-simulation process receives a SETMODE function 11 or SETPARAM function 3 message, the message includes a 16-bit parameter value (sysmsg[2] in the Setmode message, sysmsg[4] in the Setparam message) that indicates how to set BREAK ownership, as follows:

If value is	**then BREAK should be**
0	disabled and there is no owner
positive	enabled and the sender of the message is the owner
negative	enabled and the owner is indicated by the negative value

If the sender of the message is the BREAK owner, the process handle of the owner can be determined by a call to the FILE_GETRECEIVEINFO_ procedure. That process handle should already be contained in the table of openers that a terminal-simulation process typically keeps. The terminal-simulation process can remember the BREAK owner by storing its opener-table index. (For a fuller discussion of opener tables, see Section 22, Writing a Server Program.)

When requested, the terminal-simulation process must reply to the Setmode or Setparam message, including in the response a value that represents the previous owner. (See Processing System Messages, earlier in this section.) If there was no previous owner, it sends a value of 0. If there was a previous owner, it sends an internally defined value, typically the negative of the opener-table index for that owner. That way, if the negative value comes back to the terminal-simulation process in a later Setmode or Setparam message, the BREAK owner that it represents will be understood.

Basing Interprocess I/O on BREAK Mode

The terminal-simulation process must base its communication with requesters on whether the terminal-simulation process is currently in BREAK mode or normal mode:

- If the terminal-simulation process is in normal mode, the process can communicate with any requester.
- If the terminal-simulation process is in BREAK mode, the process must communicate only with a process that has BREAK access. An error must be returned to all other processes that attempt to communicate with the terminal-simulation process. This error can be returned using the `error-return` parameter of the REPLY procedure.

When the terminal-simulation process receives a SETMODE function 12 message, the process should update a variable to indicate whether the process is in BREAK mode or normal mode or update a variable indicating whether the requester has BREAK access.

When the user presses the BREAK key, the terminal simulator might enter BREAK mode, depending on the value of the *param2* parameter on the last call to SETMODE function 11 or word 1 of the *param-array* parameter passed to the last call to SETPARAM function 3.

Sending Break-on-Device Messages

The terminal-simulation process can send Break-on-device messages to requesters. To simulate a terminal accurately, the terminal-simulation process should send Break-on-device messages only to the current BREAK key owner.

To send a Break-on-device message to another process, call the BREAKMESSAGE_SEND_ procedure. To do this, you need to identify the requester process by its process handle, as well as give the file number by which the requester has the terminal-simulation process open. Both the process handle and the file number are available using FILE_GETRECEIVEINFO_ on receipt of the SETPARAM function 3 message or SETMODE function 11 message. For example:

```
PROCESS^HANDLE ':=' RECEIVE^INFO[0] FOR 10;
FILE^NUM := RECEIVE^INFO[3];
ERROR := BREAKMESSAGE_SEND_(PROCESS^HANDLE,FILE^NUM);
```

This procedure sends a Break-on-device message to the process whose process handle is stored in PROCESS^HANDLE.

The Break-on-device message sent by this procedure has the following format:

Structure of Break-on-device message (system message -105):

```
sysmsg[0] = -105
sysmsg[1] = file number of receiver’s open file to the
            terminal that indicated BREAK
sysmsg[2] = first BREAK tag word
sysmsg[3] = second BREAK tag word
```

For a complete description of the BREAKMESSAGE_SEND_ procedure, see the *Guardian Procedure Calls Reference Manual*.

25

Debugging, Trap Handling, and Signal Handling

This section deals with fixing problems that might occur with compiled code. Such problems typically are coding errors that produce unexpected results or prevent the program from continuing to execute. Coding errors are not the only possible cause of problems: a lack of a system resource (such as memory) can also prevent normal process execution. Such situations are reported as traps to TNS and accelerated programs and as signals to native programs.

This section describes the following features of the operating system:

- How to invoke a debugging program on your application
- How to deal with trap conditions (TNS and accelerated programs only)
- How to deal with signals (native programs only)

Invoking a Debugger

When your program enters the Debug state (seeSection 16, Creating and Managing Processes, for a discussion of process states) in the TNS and TNS/R environments, it runs under the control of one of two debugging programs: Debug or the Inspect program. Debug is a low-level debugger that provides bit-level information and control of registers and memory locations. The Inspect program is a symbolic debugger that allows you to control program execution, set breakpoints, examine program variables, and modify values in terms of source-language symbols.

In the TNS/R and TNS/E environments you can also use Visual Inspect to debug processes. Visual Inspect is a GUI-based native symbolic debugger that runs on a Windows workstation connected to a TNS/R or TNS/E native host. It is designed to debug native processes on a native host.

H-series and J-series systems provide two debuggers for native mode debugging in the TNS/E environment: Native Inspect and Visual Inspect. The Inspect debugger is available on both H-series and J-series systems, but can be used only for debugging TNS processes.

Visual Inspect is the primary application debugging tool in the TNS/E native environment. The H-series and J-series version of Visual Inspect debugger includes the same functionality as the G-series version. Visual Inspect supports high-level symbolic debugging of native and TNS processes (interpreted and accelerated) through a PC-based graphical user interface. Visual Inspect can also be used for debugging TNS, TNS/R, and TNS/E snapshot files.

The H-series and J-series Visual Inspect facility has been enhanced for machine-level debugging; you can now use it for low-level debugging tasks on the PC that needed to be done in previous RVUs on the NonStop server using Debug or Inspect. The following machine-level features are supported:

- Set instruction breakpoints
- Display instruction code
- Step by instruction
- Display and modify data using a numeric (non-symbolic) address
- Modify, format, and monitor registers
- Display and format data buffers as SPI or EMS buffers

These capabilities enable you to perform low-level debugging tasks that formerly needed to be done on the NonStop server using Debug or Inspect.

For operational details on Debug, see the *Debug Manual*. For operational details on the Inspect program, see the *Inspect Manual* . For operational details on Visual Inspect, see the Visual Inspect online help. For operational details on the Native Inspect program, see the *Native Inspect Manual*. This discussion is limited to how to invoke debugging on a process.

Native Inspect is a command-line symbolic debugging tool that can be used for debugging TNS/E native processes and snapshot files. It can be used for source statement level debugging as well as machine-level debugging. Native Inspect is intended as a replacement for the G-series Debug facility and the G-series Inspect debugger for native mode debugging. The command name for Native Inspect is `eInspect.`

Note. Native Inspect cannot currently be used to debug COBOL programs. The only debugger available to H-series and J-series COBOL programs is Visual Inspect.

Native Inspect provides most of the functionality of Inspect and Debug. However, the Native Inspect command syntax differs from that of Inspect and Debug. The Native Inspect syntax is based on gdb, a debugger that is widely used throughout the industry and is familiar to many application developers. In most cases, you are encouraged to use Visual Inspect as your primary application debugger. The primary advantage of Native Inspect is that it provides enhanced scripting support in the form of the Tool Command Language (TCL), a widely used scripting language, which enables you to automate many of your debugging tasks.

Note that any Inspect command files you are currently using to automate debugging operations must be converted to Native Inspect syntax.

Native Inspect supports the debugging of PIC (all TNS/E native code is PIC), whereas Inspect on the TNS/R platform cannot be used with PIC.

Getting a Process Into the Debug State

You can put a process into the Debug state in any of the following ways:

- Using the DEBUG procedure
- Using the PROCESS_DEBUG_ procedure
- Using the PROCESS_CREATE_ or PROCESS_LAUNCH_ procedure
- Using the RUND command (or the TACL run option DEBUG)
- Using the DEBUG command (from the TACL prompt)
- Using the Visual Inspect application (TNS/R and TNS/E environments only)
- Using the Native Inspect application (TNS/E environment only)
- Specifying a breakpoint

The following paragraphs describe each of the above alternatives.

Using the DEBUG Procedure

You can enter the Debug state by calling the DEBUG procedure. By calling this procedure, you invoke the debugger on your own process:

```
CALL DEBUG;
```

The calling process enters the Debug state immediately when this call is made.

Using the PROCESS_DEBUG_ Procedure

You can enter the Debug state by calling the PROCESS_DEBUG_ procedure. Using this procedure, you can invoke the debugger on your own process or on another process.

To invoke the debugger on another process, you must supply the process handle of the process you wish to debug:

```
CALL PROCESS_DEBUG_(PROCESS^HANDLE);
```

To debug another process this way, your process must either have super ID access, be the group manager of the process access ID, or be a process with the same process access ID as the target process.

You can invoke the debugger on your own process by calling the PROCESS_DEBUG_ procedure without any options. However, the preferred method of invoking the debugger on your own process is by calling DEBUG.

Using the PROCESS_CREATE_ or PROCESS_LAUNCH_ Procedure

You can run the debugger on a new process as soon as the process starts by supplying the *`debug-options`* parameter to the PROCESS_CREATE_ procedure. To run the debugger, bit 12 of this parameter must be set to 1.

You can accomplish the same thing using the PROCESS_LAUNCH_ procedure. The difference is that the debug options bits are passed to PROCESS_LAUNCH_ in a field in a structure instead of in a separate parameter.

The following example uses PROCESS_CREATE_ to start a new process in the Debug state:

```
OBJFILE ':=' "REQFILE" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR ‘-’ @OBJFILE;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEINCALL;
PROCESS^NAME ':=' "$REQ" -> @S^PTR;
PROCESS^NAME^LENGTH := @S^PTR ‘-’ @PROCESS^NAME;
DEBUG^OPTIONS := 0;
DEBUG^OPTIONS.<12> := 1;
ERROR := PROCESS_CREATE_(OBJFILE:OBJFILENAME^LENGTH,
                         !library^file:length!,
                         !swap^file:length!,
                         !ext^swap^file:length!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         PROCESS^NAME:PROCESS^NAME^LENGTH,
                         PROCESS^DESCRIPTOR:MAXLEN,
                         PROCESS^DESCRIPTOR^LENGTH,
                         !nowait^tag!,
                         !hometerm:length!,
                         !memory^pages!,
                         !job^id!,
                         !create^options!,
                         !defines:length!,
                         DEBUG^OPTIONS);
```

Using the RUND Command

You can use the RUND command to start a process in the Debug state. You must supply the name of an object file as the parameter to the RUND command.

```
RUND OBJFILE
```

To invoke the debugger, you must either have the process access ID of the process you want to debug, be the group manager of that process, or be the super ID user.

Using the RUNV Command

The RUNV command runs a process under control of Visual Inspect. The equivalent OSS command is `runv`. For example:

`>RUNV WSADDR=MYPC GPROG` (Guardian)

`$ runv -wsaddr=MYPC oss prog` (OSS)

These commands start a Visual Inspect debug session on the workstation MYPC.

Using the DEBUG Command

You can put a process into the Debug state by entering the DEBUG command. You need to supply the CPU number and PIN of the process you want to debug. Typical use of this command is as follows:

1. Run a process with the RUN command:

   ```
   > RUN OBJFILE
   ```

2. Press the BREAK key to return the command-interpreter prompt.
3. Find the CPU number and PIN of the process using the STATUS command:

   ```
   > STATUS
   Process       Pri PFR %Wt Userid Program file      Hometerm
            2,49 139   R 000 8,160  $APPLS.PROGS.OBJ  \SYS.$AA
   ```

4. Run the DEBUG command on the process:

   ```
   > DEBUG 2,49
   DEBUG 002020, 000317, UC.00
   106,02,00049-
   ```

Using the Visual Inspect Application

Visual Inspect is a NonStop operating system symbolic debugger that uses program visualization, direct manipulation, and other techniques to improve productivity in the development or production environment. It supports these machine architectures and compilers:

CISC (TNS C/C++, COBOL85, TAL)

RISC (TNS/R Native C/C++, NMCOBOL, pTAL)

Itanium® (TNS/E Native C/C++, ECOBOL, EpTAL)

Visual Inspect uses client/server architecture. The client runs on a Windows NT, Windows 95, Windows 2000, or Windows XP workstation. The server runs on a NonStop host. Working together, the client and server components allow you to control, analyze, observe and alter application execution so that you can quickly find and correct logic errors.

Refer to the Visual Inspect online help, included with the application, for further information on starting and managing a process debugging session.

Using the Native Inspect Application

Native Inspect is a symbolic command-line debugger used for debugging TNS/E native process and snapshots on HP TNS/E systems. You can use Native Inspect to debug a running process:

```
TACL> debug $myproc
```

This invokes Native Inspect on the home terminal of process $myproc.

You can also specify a home terminal on which you want Native Inspect to run:

```
TACL> debug $myuproc, term $ztn10.#pthef
```

Native Inspect is used to debug native processes; it does not support process debugging of TNS processes. See the *Native Inspect Manual* for further information.

Specifying a Breakpoint

You can put a process into the Debug state by specifying a breakpoint, as follows:

1. With the process already in the Debug state, set a breakpoint.
2. Resume normal processing of the process.
3. The process enters the Debug state when it encounters the breakpoint.

Specifying the Debugging Environment

Specifying the debugging environment primarily involves choosing between the Debug and Inspect debuggers in the TNS and TNS/R environments, the Visual Inspect debuggers in the native environments and the Native Inspect debugger in the TNS/E environment. However, you can also choose to create a saveabend file that saves data area and file status information if the process abnormally terminates. The saveabend feature is available only with the Inspect debugger.

By default, you get the Debug program when entering the Debug state on TNS/R systems and Visual Inspect when entering the Debug state on TNS/E systems. If the parent process is running under a debugger when a new process is created, then the new process inherits the debugging environment of the parent process. However, you can specify the debugging environment as follows:

- Using compiler directives
- Using the Binder, `nld` utility (in the TNS/R environment), or the `eld` utility (in the TNS/E environment)
- Using the TACL SET INSPECT command
- Using the TACL RUN command
- Using the PROCESS_CREATE_ or PROCESS_LAUNCH_ procedure

The following paragraphs describe each of the above alternatives.

Note that once your process is in the Debug state under the Inspect program, you can switch back and forth between Inspect and Debug at will.

Note. If If Inspect of Visual Inspect is not available on your systems and you try to invoke it, your process runs under Debug (on TNS/R systems) or Native Inspect when available on TNS/E systems. If Native Inspect is not available, then the request to debug the process is ignored and the process continues execution.

Switching Debuggers During a Debugging Session

The native debuggers Inspect, Visual Inspect, and Native Inspect (in the TNS/E environment only), allow you to switch from one debugger to an other while debugging a process. The ability to switch debuggers is especially useful if a process enters the debug state under control of an inappropriate debugger; for example, if a TNS program automatically enters the debug state under control of Native Inspect. Since Native Inspect does not support TNS mode debugging, you could use this feature to switch to Inspect. See the *Inspect Manual,* the *Native Inspect Manual*, and the Visual Inspect online help for details.

Using Compiler Directives

If you specify the INSPECT compiler directive in your TAL or pTAL program, then the process automatically runs under the Inspect program whenever the process enters the Debug state.

The SAVEABEND compiler directive in your TAL or pTAL program also implies that the Inspect program is the debugger, as well as specifies a saveabend file for the process.

See the *TACL Reference Manual* for details on these compiler directives.

Some other programming languages offer equivalent capabilities. Refer to the appropriate programming language manuals.

Using the Binder and `nld` or `eld` Utilities

The Binder and the `nld` utility (in the TNS/R environment) and the `eld` utility (in the TNS/E environment) also have commands that allow you to select the Inspect program (in the TNS/R environment) or the Native Inspect program (in the TNS/E environment) while linking. The SET INSPECT ON command, in either utility, causes your process to automatically run under the Inspect program on TNS/R systems. On TNS/E systems, SET INSPECT ON sets Visual Inspect as the default debugger.

The SET SAVEABEND ON command also implies the Inspect program and specifies a saveabend file for the process.

See the *Binder Manual* for details on these Binder commands. See the *nld Manual* for information about using the `nld` utility in the TNS/R environment. See the *eld Manual* for information about using the eld utility in the TNS/E environment.

Using the TACL SET INSPECT Command

You can use the TACL SET INSPECT command to establish the debugging environment for processes that will be started by the current TACL process. The following example establishes the Inspect program as the default debugger:

```
> SET INSPECT ON
```

The next example also establishes the Inspect program as the default debugger while specifying that a saveabend file should be created:

```
> SET INSPECT SAVEABEND
```

The last example establishes Debug as the default debugger:

```
> SET INSPECT OFF
```

The environment set up by the SET INSPECT command can be overridden by a RUN command, a Binder command, or a TAL compiler directive.

See the *TACL Reference Manual* for a complete description of the SET INSPECT command.

Using the TACL RUN Command

You can use the TACL RUN command to set the debugging environment for the process that you are starting by supplying the Inspect option.

You can establish the Inspect program as the debugger as follows:

```
> RUN objfile /INSPECT ON/
```

You can specify a saveabend file and establish the Inspect program as the debugger as follows:

```
> RUN objfile /INSPECT SAVEABEND/
```

To select Debug as the debugger, enter the following:

```
> RUN objfile /INSPECT OFF/
```

The environment set up by the RUN command can be overridden by a linker command (with either Binder or nld) or a compiler directive.

For more information about the TACL RUN command, refer to the *TACL Reference Manual*.

Using the PROCESS_CREATE_ or PROCESS_LAUNCH_ Procedure

You can specify the process debugging environment when you create a process using the PROCESS_CREATE_ procedure. You have already seen how to use this procedure to put the new process in the Debug state. Now, using bit 15 of the *`debug-options`* parameter, you can specify the debugger you want to use.

Set bit 15 of the *`debug-options`* parameter to 1 to use the Inspect program, and set it to 0 to use Debug. Whether the setting of bit 15 overrides any value supplied by the linker or the compiler depends on bit 14: if bit 14 is set to 1, then the value supplied in bit 15 specifies the debugger, regardless of any value supplied by the linker or compiler. If bit 14 is 0, then bit 15 does not override any value supplied by the linker or compiler.

To specify a saveabend file for the process, set bit 13 to 1.

You can accomplish the same things using the PROCESS_LAUNCH_ procedure. The difference is that the debug options bits are passed to PROCESS_LAUNCH_ in a field in a structure instead of in a separate parameter.

The following example uses PROCESS_CREATE_ to specify the Inspect program, regardless of any value supplied by the linker or compiler:

```
OBJFILE ':=' "REQFILE" -> @S^PTR;
OBJFILENAME^LENGTH := @S^PTR '-' @OBJFILE;
NAME^OPTION := ZSYS^VAL^PCREATOPT^NAMEINCALL;
PROCESS^NAME ':=' "$REQ" -> @S^PTR;
PROCESS^NAME^LENGTH := @S^PTR '-' @PROCESS^NAME;
DEBUG^OPTIONS := 0;
DEBUG^OPTIONS.<13> := 1;
DEBUG^OPTIONS.<14> := 1;
DEBUG^OPTIONS.<15> := 1;
ERROR := PROCESS_CREATE_(OBJFILE:OBJFILENAME^LENGTH,
                         !library^file:length!,
                         !swap^file:length!,
                         !ext^swap^file:length!,
                         !priority!,
                         !processor!,
                         PROCESS^HANDLE,
                         !error^detail!,
                         NAME^OPTION,
                         PROCESS^NAME:PROCESS^NAME^LENGTH,
                         PROCESS^DESCRIPTOR:MAXLEN,
                         PROCESS^DESCRIPTOR^LENGTH,
                         !nowait^tag!,
                         !hometerm:length!,
                         !memory^pages!,
                         !job^id!,
                         !create^options!,
                         !defines:length!,
                         DEBUG^OPTIONS);
```

Handling Trap Conditions

During program execution of TNS or accelerated programs, all error and exception conditions not related to input or output are handled by a returned error parameter value. Conditions that are trapped typically are caused by coding errors in your application program or by a shortage of resources. For example, errors such as “arithmetic overflow” might be caused by erroneous application code, whereas an error such as “no memory available” might originate from the memory manager.

Table 25-1 provides a summary of the conditions that can cause your process to trap. Each trap condition is identified by a trap number.

Table 25-1. Summary of Trap Conditions

Trap Number	Cause of Trap
0	Invalid address reference
1	Instruction failure
2	Arithmetic overflow
3	Stack overflow
4	Process loop-timer timeout
5	Invalid call from process with PIN greater than 255
8	Signal (Under very unusual circumstances, a signal is delivered to a TNS process and appears as a trap 8.)
11	Memory manager disk read error
12	No memory available
13	Uncorrectable memory error

Refer to the *Guardian Procedure Errors and Messages Manual* for a detailed description of each trap condition, including what might have caused the trap and recommended action.

You can respond to a trap in one of the following ways:

- In the TNS/E environment all traps result in the process abending. This is the default action. If this is a TNS process, debug it using INSPECT; the TNS/E environment does not recognize Native Inspect for TNS processes.
- The TNS or TNS/R environments allow either the Inspect program or Debug to be automatically invoked. This is the default action. If Inspect is specified for the process and it is available, then Inspect is the default debugger; otherwise Debug is the default debugger. In other words, if you do nothing, all traps result in control being passed to either Debug or the Inspect program. A screen similar to the following appears:

  ```
  >RUN Z
  INSPECT-Symbolic Debugger-T9673C20-(10 July89) System \SYS
  Copyright Tandem Computers Incorporated 1983, 1985-1989
  INSPECT TRAP 2- (arithmetic overflow)
  099,07,053 #TRAP^USER.#43(MYPROG)
  -Z-
  ```

  Note that line 3 of the display identifies the trap condition.

  Refer to the *Inspect Manual* and the *Debug Manual* for operational details on the Inspect program and Debug.

- Choose to handle traps with your own trap-handling code. The ARMTRAP procedure allows you to specify the address of your trap handler.

  The rest of this section describes how to write your own trap handler.

- Disable all trap handlers, including Inspect and Debug. In this case, your process abnormally terminates if a trap condition occurs. The ARMTRAP procedure allows you to disable traps.

Note. HP recommends either writing your own trap handler or disabling traps. We make this recommendation because a typical user does not know how to respond to an application that goes into Inspect or Debug mode. Moreover, the operator’s console does not receive a message to indicate that the process has trapped. Hence the process hangs in Inspect or Debug. Stopping the process in the trap handler or disabling traps prevents the application from hanging. The trap handler should also be written to inform the console operator of the problem.

Trap conditions that occur when user code is being processed cause an immediate trap. If a trap condition should occur when system code is being executed, then the trap does not occur until control returns to the user code.

Setting Up a Trap Handler

When setting up a handler, you use the ARMTRAP procedure to perform two functions:

- Set up a pointer to the start of the trap-handling code. This location will be the entry point into the trap handler when a trap occurs.
- Specify the start of the trap handler local data area. You typically locate this area at the high end of the user data stack, where it is less likely to compete with the application for stack space.

The following code fragment performs the two functions described above:

```
TRAPHANDLR^ADDR := @TRAP;
TRAPSTACK^ADDR := $LMIN(LASTADDR, %77777) - 500;
CALL ARMTRAP(TRAPHANDLR^ADDR,
             TRAPSTACK^ADDR);
```

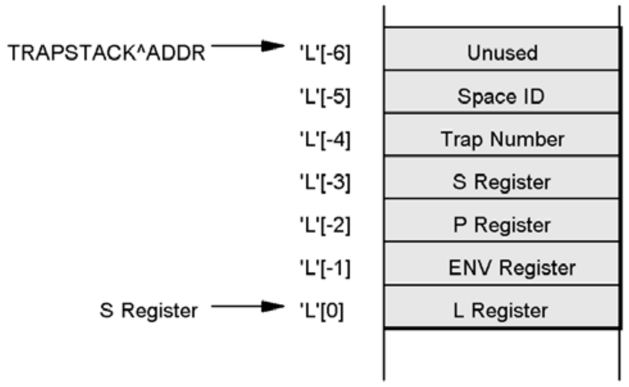
Here, “@TRAP” identifies the start of the trap code as the following label:

```
TRAP:
```

The expression “$LMIN(LASTADDR,%77777) - 500” indicates that the start of the trap-handler local data area will be 500 locations before the end of the user data area or 500 locations before location %77777, whichever is the lesser.

When a trap occurs, the first six words of the trap handler local data area will contain environment information as well as the trap number itself. The S (stack pointer) and L (local data pointer) registers will point to the start of the local data area plus six. Figure 25-1 shows the allocation of stack space to the trap handler.

Figure 25-1. Trap Handler Data Stack When Trap Occurs



VST115.VSD

Processing a Trap

The code for processing a trap depends on what you want to do for a given trap condition. See Writing a Trap Handler: Examples at the end of this section for an example of how to process a trap due to arithmetic overflow.

However, if you want to return to the application after processing the trap, then you need to save the stack registers immediately on entering the trap-handling code. When the trap-handling code is entered, the stack registers contain information at the point that the trap occurred. Before the trap handler changes the register values, you should save them so that you can restore them when you exit the trap handler.

To save the stack registers, you must push them onto the data stack using the PUSH machine instruction as follows:

```
CODE (PUSH %777);
```

The parameter %777 causes all eight stack registers to be saved.

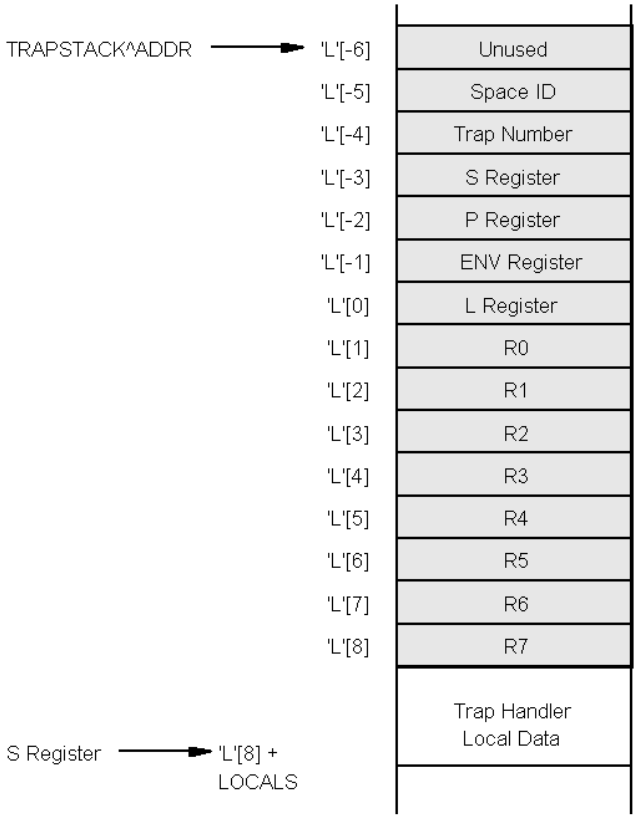
You must then allocate any storage you need for local variables by advancing the S register by the number of words needed to save the local data. Use the ADDS machine instruction as follows.

```
CODE (ADDS LOCALS);
```

Here, “LOCALS” specifies the number of two-byte words of local data.

The trap stack space now has data space allocated as shown in Figure 25-2.

Figure 25-2. Trap Handler Data Stack After Storage Allocation


TRAPSTACK^ADDR
'L'[-6]
Unused
'L'[-5]
Space ID
'L'[-4]
Trap Number
'L'[-3]
S Register
'L'[-2]
P Register
'L'[-1]
ENV Register
'L'[0]
L Register
'L'[1]
R0
'L'[2]
R1
'L'[3]
R2
'L'[4]
R3
'L'[5]
R4
'L'[6]
R5
'L'[7]
R6
'L'[8]
R7
Trap Handler
Local Data
S Register
'L'[8] +
LOCALS

VST116.VSD

Exiting a Trap Handler

Once you have processed the trap condition, you might want to exit from your trap handler and return to your application. To do this, you need to reissue the ARMTRAP procedure call with the *`traphandlr-addr`* parameter set to zero:

```
TRAPHANDLR^ADDR := 0;
TRAPSTACK^ADDR := $LMIN(LASTADDR, %77777) - 500;
CALL ARMTRAP(TRAPHANDLR^ADDR,
             TRAPSTACK^ADDR);
```

Calling ARMTRAP in this way restores all register values (including the stack registers) to what they were when the trap occurred. The ARMTRAP procedure uses the values saved in locations ‘L’ [-6] through ‘L’ [8] to achieve this. This call to ARMTRAP normally rearms the trap handler using the same *`traphandlr-addr`*. Refer to the description of ARMTRAP in the *Guardian Procedure Calls Reference Manual* for details.

Exiting After an Arithmetic Overflow Trap Condition

If the cause of the trap was an arithmetic overflow condition, then you must reset the arithmetic overflow bit in the ENV register (bit 10) before exiting the trap handler.

Exiting After a System Code Trap

Another case to be aware of is when exiting the trap handler after a trap condition that occurred when system code was being executed. Here, when control is passed to the trap handler, the location of the trap passed to the trap handler is the location of the call to the system procedure; the S register at the trap contains -1 to signify that a deferred trap occurred while in system code. Therefore, your process cannot resume execution at the point of the trap following a trap in system code because the correct value for the S register is lost. Your program can, however, exit elsewhere. See Exiting to Another Destination later in this section.

The only trap condition in system code that you can resume after is the process loop-timer trap. If a process loop-timer times out and causes a trap while in system code, the trap is deferred until the process returns to the user environment but the value of the S register at this trap is not -1. In this case, the S register contains the correct value, which allows the process to easily resume execution following a loop-timer timeout.

Exiting to Another Destination

A trap handler can exit to some other place in the program, such as a restart point, by putting the appropriate values into various locations in the region of ‘L’[-5] through ‘L’[8]. The code location is specified in the space index, ENV, and P values. The stack location is specified in S and L values. Typically, a restart point is at a label at the beginning of a statement, so the registers are empty: set RP to 7 in ENV.<13:15> (‘L’[1:8] are immaterial).

Disabling Trap Handling

You can use the ARMTRAP procedure to disable all trap handlers, including Inspect and Debug, in addition to user-written trap handlers. Doing so causes your process to abnormally terminate if a trap condition occurs. In the TNS/E environment, the ARMTRAP procedure disables all trap handlers as the default behavior.

To disable trap handling, set the trap stack address in a call to the ARMTRAP procedure to a negative number:

```
CALL ARMTRAP(-1,-1);
```

Trap Handling on Native Systems

Special restrictions apply to trap handlers that execute on native systems. You should observe the following rules:

- Trap P variable

  The TNS trap P variable is only approximate for a process running in accelerated mode. You should not use it to inspect the code area and determine the failing instruction.

  You should not increment the trap P variable and resume execution; doing so causes undefined results. However, you can change the trap P variable to a valid TNS restart point. See Exiting to Another Destination earlier in this section.

- Invalid trap ENV fields

  ENV.RP is not valid for a process running in accelerated mode. For compatibility with TNS mode and TNS CPUs, programs must set ENV.RP to an appropriate value before resuming; to resume at a statement, the appropriate value is usually 7.

  The ENV fields N, Z, and K are not reliable for a process running in accelerated mode.

- Register stack R[0:7]

  The contents of the TNS register stack are not valid in accelerated mode and are not dependable in TNS mode. You should never change the register stack when attempting to resume at the point of the trap.

- Functions

  A trap handler procedure must not be a function that returns a value.

- Resuming from a trap

  The following ways of resuming execution from a trap are supported:

  - Clear the overflow bit in the trap ENV variable and resume from a trap 2 (arithmetic overflow).
  - Resume after a loop timer interrupt (trap 4).
  - Jump to a restart point by changing the trap variables P, L, ENV, space ID, and S. See Exiting to Another Destination earlier in this section.
  - Terminate the process.

Writing a Trap Handler: Examples

The following program shows an example of a trap handler that displays the contents of the P register when an arithmetic overflow occurs. After displaying the P register, the trap handler returns to the application.

If any trap condition other than arithmetic overflow occurs, then the trap handler calls the DEBUG procedure.

The example consists of two procedures:

- The OVERFLOWTRAP procedure sets up the trap handler with a call to ARMTRAP and also provides the code for the trap handler itself.
- The TRAP^USER procedure is the main procedure. It calls the OVERFLOWTRAP procedure to set up the trap handler, then causes an arithmetic overflow trap condition by attempting to divide by zero.

```
?INSPECT,SYMBOLS

!Global variables:
INT TERM^NAME[0:11];
INT TERM^NUM;

?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(INITIALIZER,PROCESS_GETINFO_,
?  FILE_OPEN_,ARMTRAP,WRITE,DEBUG,NUMOUT,LASTADDR)
?LIST


!------------------------------------------------------------
! Sets up the start address of the trap handler and the stack
! space for the trap handler by calling the ARMTRAP
! procedure.  This procedure also supplies the code for the
! trap handler, including a second call to ARMTRAP that exits
! the trap handler.
!------------------------------------------------------------

PROC OVERFLOWTRAP;
BEGIN
   INT REGS = 'L' +1,           !R0 to R7 saved here
       WBUF = 'L' +9,           !buffer for terminal I/O
       PREG = 'L' -2,           !P register at time of trap
       EREG = 'L' -1,           !ENV register at time of trap
       TRAPNUM = 'L' -4,        !trap number
       SPACEID = 'L' -5;        !space ID of trap location
       DEFINE OVERFLOW = <10>#;!overflow bit in ENV register
       STRING SBUF = WBUF;      !string overlay for I/O buffer
       LITERAL LOCALS = 15;     !number of words of local
                                ! storage

!  Arm the trap:

   CALL ARMTRAP(@TRAP, $LMIN(LASTADDR,%77777) - 500);
   RETURN;
```

```
!  Enter here on trap:

   TRAP:

!  Save registers R0 through R7 and allocate local storage:

   CODE(PUSH %777; ADDS LOCALS);

!  Call DEBUG if trap not an arithmetic overflow
!  condition:

   IF TRAPNUM <> 2 THEN CALL DEBUG;

!  Format and print the message on the home terminal with
!  the P register value displayed in octal:

   SBUF ':=' "ARITHMETIC OVERFLOW AT %";
   CALL NUMOUT(SBUF[24], PREG, 8, 6);
   CALL WRITE(TERM^NUM,WBUF,30);
   IF <> THEN CALL DEBUG;

!  Clear the overflow bit in the ENV register:

   EREG.OVERFLOW := 0;

!  Exit the trap handler and restore old values of
!  registers:

   CALL ARMTRAP(0, $LMIN(LASTADDR,%77777) - 500);
END;


!------------------------------------------------------------
! Main procedure reads Startup message, calls OVERFLOWTRAP,
! opens the home terminal, and creates an arithmetic overflow
! condition by attempting to divide by zero.
!------------------------------------------------------------

PROC TRAP^USER MAIN;
BEGIN
   STRING .TERM^NAME[0:MAXLEN - 1];
   INT     I, J, LEN;

!  Read the Startup message:

   CALL INITIALIZER;

!  Call the OVERFLOWTRAP procedure to arm the trap:

   CALL OVERFLOWTRAP;
```

```
!  Open home terminal:

   CALL PROCESS_GETINFO_(!process^handle!,
                         !file^name!,
                         !file^name^len!,
                         !priority!,
                         !moms^processhandle!,
                         TERM^NAME:MAXLEN,
                         LEN);

   CALL FILE_OPEN_(TERM^NAME:LEN,TERM^NUM);
   IF <> THEN CALL DEBUG;

!  Set up an arithmetic overflow condition to cause the
!  trap:

   J := 0;
   I := I/J;

END;
```

In the following example, a trap causes the current code sequence to be abandoned and an alternate code sequence executed instead. Several points are illustrated:

- The procedure TRAP_GUARD sets its caller as the destination (the code address and stack environment) to be used if a trap occurs; it also arms the trap handler.
- When a trap occurs, the information about the trap is saved globally (for possible display or analysis).
- The trap handler exits, and control is returned to the designated destination. Consistent values are supplied for space ID, S, P, ENV, and L. ENV.RP is set to 0 and RP[0] is set to a specific value so that returning from the trap handler effects a return from the TRAP_GUARD function with the result equal to TRUE (-1).
- Incidentally, the trap handler stack is allocated among the globals (below the normal stack) instead of at the top of the stack.

```
! Global declarations for trap handling
INT saved_L, ! destination when trap occurs: L,
    saved_E, !   ENV (stack-marker form, with space index),
    saved_P, !   P,
    saved_S; !   S
STRUCT trapframe_template(*);
  BEGIN ! stack frame layout of trap handler
    INT spaceid,trapnum,S,P,E,L,R[0:-1];
  END;
STRUCT .trapstate(trapframe_template); ! data from most
                                       ! recent trap
INT .trap_stack[0:499]; ! space for trap handler stack

?PUSHLIST,NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS(ARMTRAP,ABEND)
?POPLIST

INT PROC TRAP_GUARD; ! procedure to set trap destination,
BEGIN                ! arm trap handler, and handle traps
  STRUCT tf(trapframe_template) = 'L'-5;
  INT L = 'L';

  saved_L := L;      ! Save the
  saved_E := L[-1]; !   destination
  saved_P := L[-2]; !     code and stack
  saved_S := @L-3;  !       locations

  CALL ARMTRAP(@traphandler,@trap_stack); ! Arm the
                                          !   trap handler
  RETURN 0; ! Return False: no trap
  ! If a trap occurs subsequently, TRAP_GUARD will return
  ! again to the site of the last call, but this time the
  ! value will be True.

traphandler: ! code invoked by system when trap occurs

  CODE(PUSH %777);  ! Save all stack registers

  trapstate ':=' tf FOR 1 ELEMENTS; ! Save trap state

  ! Set state to return again from last call to TRAP_GUARD
  tf.spaceid := saved_E; ! Only space index is signficant
  tf.S := saved_S;
  tf.E := saved_E LAND $COMP(%37); ! Clear out space index:
                        !   CC = 0 to ensure legal value
                        !   RP = 0 to return one item in R[0]
  tf.P := saved_P;
  tf.L := saved_L;
  tf.R[0] := -1; ! Return True from TRAP_GUARD
  CALL ARMTRAP(0,@trap_stack); ! Exit from trap handler
END;  ! TRAP_GUARD
```

```
! One or more sequences like the following
! surround code that might trap.

  IF TRAP_GUARD THEN ! TRAP_GUARD will later return True
    BEGIN            ! to execute this THEN clause, if a
                     ! trap occurs in the ELSE clause.
    .
    . ! code to handle the trap contingency
    .
    END
  ELSE               ! TRAP_GUARD initially returns False
    BEGIN            ! to execute this ELSE clause.
    .
    . ! code that might trap
    .
    END;
  CALL ARMTRAP(-1,-1); ! Abend if trap where unexpected
  ! (Note that if the procedure that called TRAP_GUARD were
  ! to exit, and a trap occurred while traps were still
  ! armed, the saved destination would be wrong.)

  .
  . ! code where no trap is tolerated
  .
```

Handling Signals

Native Guardian processes receive signals when run-time events occur that require immediate attention; they cannot receive traps. (OSS processes, both TNS and native, receive signals.) Signals are software interrupts that provide a way of handling asynchronous events, such as a timer expiration, detection of a hardware fault, abnormal termination of a process, a lack of system resources, a process sending a signal to itself, or any trap condition normally detectable by a TNS process.

The signals facility is unchanged in the H-series. However, the H-series signals facility uses the H-series debuggers, as shown in the following table:

CPU and Execution Mode	**Debuggers**
TNS/R native mode	Visual Inspect, Inspect, Debug
TNS/E native mode	Visual Inspect, Native Inspect

The details mentioned for H-series are also applicable to J-series RVUs.

See the *Inspect Manual,* the *Native Inspect Manual*, and the Visual Inspect online help for details.

Programs running as native Guardian processes can use the following functions and procedures to receive and handle signals:

- Signals functions in the POSIX.1 standard. These are the signals functions provided in the Open System Services (OSS) application program interface (API). These functions are all available in C and most are available in pTAL.
- HP signals extensions to the POSIX.1 standard. These procedures are written especially for applications that focus on handling signals indicating conditions known as traps in TNS processes. These procedures are available in pTAL and in C.

About Signals

The following brief discussion provides basic POSIX.1 signal concepts and terminology. It is intended to provide you with a framework to understand how to handle signals in native Guardian processes. For more conceptual information, see commercial texts on UNIX programming. For the specifics of the signals functions in the OSS API, see the *Open System Services System Calls Reference Manual* and the *Open System Services Library Calls Reference Manual*. For the specifics of the procedures in the Guardian API, see the *Guardian Procedure Calls Reference Manual*.

Signal Generation, Delivery, and Actions

A signal is generated for a process when the event that causes the signal occurs. When a signal is delivered to a process, an action for the signal is taken. During the time between generation and delivery of a signal, the signal is pending.

A process can select one of three standard actions for most signals:

- Let the default action apply. For most signals, the default action is to terminate the process.
- Ignore the signal. Delivery of the signal has no effect on the process. Some signals, such as `SIGSTOP`, cannot be ignored.
- Catch the signal. You supply a signal-catching function called a signal handler that contains instructions to be executed when a particular signal occurs. Some signals, such as `SIGABEND`, cannot be caught.

Blocking Signals

Each process has a signal mask that defines the set of signals currently blocked from delivery to it. Signals that cannot be ignored cannot be blocked from delivery to a process.

If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal remains pending until the process either unblocks the signal or changes the action to ignore the signal.

If a blocked signal is generated for a process more than once before the process unblocks the signal, it is discarded and only one instance of the signal remains pending. The order in which pending signals are delivered to a process once they are unblocked is unspecified.

Default Signal Settings in Guardian Processes

When a Guardian native process is created, the process signal mask is initialized so that no signal is blocked. If a signal is delivered to the process for which the default action is to terminate the process, process termination remains the default action. For any other signal, the default action is set to ignore the signal.

Comparing Traps and Signals

In the native Guardian environment, a signal is said to be nondeferrable if it cannot be blocked and cannot be ignored. Whether action upon a signal can be deferred depends not on the type of signal it is but on whether the signal is generated by the system, by the process itself (to itself) using the `raise()` function, or by a timer to the process.

A signal generated by the system to indicate a run-time error in the process cannot be deferred. A signal generated by a process to itself or by a timer can be deferred.

A trap received by a TNS process is equivalent to a system-generated nondeferrable signal received by a native process. Traps are a subset of POSIX.1 signals. Table 25-2 shows the signals that map to traps.

Table 25-2. Map of Signals to Traps

Signal	Description	Trap Number
SIGABRT	Abnormal termination	None: new
SIGFPE	Arithmetic overflow	2
SIGILL	Invalid hardware instruction	1
SIGLIMIT	Limits exceeded	5
SIGMEMERR	Uncorrectable memory error	13
SIGMEMMGR	Memory manager disk read error	11
SIGNOMEM	No memory available	12
SIGSEGV	Invalid memory reference	0
SIGSTK	Stack overflow	3
SIGTIMEOUT	Process loop timeout	4

Note. Under very unusual circumstances, a signal can be delivered to a TNS Guardian process. Any signal is delivered as trap number 8 to a TNS Guardian process.

When Would You Use a Signal Handler?

If your applications have trap handlers to handle trap conditions in TNS processes, you should write signal handlers to handle the equivalent signals in native processes.

Signal handlers are also helpful in applications that must run all the time with minimal, if any, operator intervention, such as a print spooler. A signal handler catches signals that might cause the spooler process to terminate. Having a signal handler can prevent the spooler process from terminating and waiting to be restarted by an operator.

Default Signal Handlers

If what you need to do is display process context information before taking the default action for a signal and the program is in C, consider using the default signal handler provided by the Common Run-Time Environment (CRE). The CRE sets up a signal handler for all signals in which the default action is not to ignore the signal.

If a C program has not set up its own signal handler, the CRE signal handler takes over when a signal is delivered to the program. The CRE signal handler displays a diagnostic message explaining the signal and also displays a stack trace. If the default action for the signal was to terminate the process, the CRE signal handler calls the PROCESS_STOP_ procedure.

The CRE does not provide a default signal handler for programs written in pTAL. To display process context information when a pTAL process receives a signal, you can use the HIST_INIT_, HIST_FORMAT_, and HIST_GETPRIOR_ procedures.

Standard Signals Functions

Table 25-3 shows the standard signals functions that native processes (Guardian or OSS) can use to handle signals. Notice that native Guardian processes cannot send or receive a signal from another process using the `kill()` function.

Table 25-3. Signals Functions That Conform to the POSIX.1 Standard (page 1 of 2)

C Function	pTAL Procedure	Description
`abort()`	No equivalent	Terminates the calling process by sending it a `SIGABRT` signal.
`alarm()`	No equivalent	Sets or changes a timer that expires at a specified time in the future. When the timer expires, the `SIGALRM` signal is generated.
`kill()`	No equivalent	Sends a signal to a process. The `kill()` function requires an OSS process ID to identify the process receiving the signal, and a Guardian process does not have an OSS process ID. A native Guardian process cannot receive a signal from or send a signal to another process (Guardian or OSS) using the `kill()` function.
`longjmp()`	LONGJMP_	Performs a nonlocal goto. Restores the execution context saved by a call to the `setjmp()` function.
`pause()`	No equivalent	Suspends the calling process until it receives a signal whose action is either to execute a signal handler or to terminate the process.
`raise()`	RAISE_	Sends a signal to the calling process.
`setjmp()`	SETJMP_	Saves the current execution context, which is restored after a call to `longjmp()`.
`sigaction()`	SIGACTION_	Specifies the action to be taken upon delivery of a signal. An action for a signal remains in effect until it is changed by another call to `sigaction()`.
`sigaddset()`	SIGADDSET_	Adds a signal to a signal set (not to a process signal mask).
`sigdelset()`	SIGDELSET_	Deletes a signal from a signal set (not from a process signal mask).
`sigemptyset()`	SIGEMPTYSET_	Initializes a signal set (not a process signal mask) to exclude all signals.
`sigfillset()`	SIGFILLSET_	Initializes a signal set (not a process signal mask) to include all signals.
`sigismember()`	SIGISMEMBER_	Tests whether a signal is a member of a signal set (not of a process signal mask).

Table 25-3. Signals Functions That Conform to the POSIX.1 Standard (page 2 of 2)

C Function	pTAL Procedure	Description
`siglongjmp()`	SIGLONGJMP_	Performs a nonlocal goto. It is often called from a signal handler to return to the main loop of a program instead of returning from the handler. It restores the execution context saved by a call to the `sigsetjmp()` function, including the process signal mask if it was saved in a `sigsetjmp()` call.
`signal()`	SIGNAL_	Specifies the action to be taken upon delivery of a signal. A signal action specified by this function is reset to the default action each time the signal is delivered.
`sigpending()`	SIGPENDING_	Returns the set of signals that are blocked from delivery and pending to the calling process.
`sigprocmask()`	SIGPROCMASK_	Changes or examines a process signal mask.
`sigsetjmp()`	SIGSETJMP_	Saves the current execution context, which can include the process signal mask, and is restored after a call to `siglongjmp()`.
`sigsuspend()`	SIGSUSPEND_	Changes a process signal mask and suspends the calling process until either a signal is caught or a signal occurs that terminates the process.
`sleep()`	No equivalent	Suspends the calling process for a specified period of time until either the time elapses, a signal is caught, or a signal occurs that terminates the process.

Using Standard Signals Functions

There are many ways to use the standard signals functions in your application programs. For information about writing standard, portable signal handlers, see commercial texts on UNIX programming. The following discussion provides considerations for using some of the standard signals functions and the sequence in which you might use them.

Tailoring the Signal Mask

The signal mask of a process contains the signals to be blocked from delivery to a process. Remember that a process signal mask cannot include signals that cannot be ignored. You can construct a signal set with the `sigaddset()`, `sigdelset()`, `sigemptyset()`, and `sigfillset()` functions. This signal set is essentially a draft of a process signal mask, but the signal set does not become an official process signal mask until the set is passed to the `sigprocmask()`, `sigpending()`, or `sigaction()` function. These functions validate the content of the process signal mask before installing it.

Before executing a signal handler, the new process signal mask is installed. This mask is the union of the current process signal mask and the signal being delivered.

Specifying an Action for a Signal

A process also uses the `sigaction()` function to specify the action to be taken in response to a signal. The action can be to ignore the signal, take the default action for the signal, or catch the signal. If the action is to catch the signal, the `sigaction()` function installs a signal-handling function. When the signal handler finishes, if the process can continue, it resumes executing where it left off before the signal was delivered.

Resuming a Process in a Different Context

A process can resume in a different context by using a combination of the `sigsetjmp()` and `siglongjmp()` functions. `sigsetjmp()` and `siglongjmp()` also allow the process to save and restore the state of the process signal mask before the call to the signal handler, whereas `setjmp()` and `longjmp()` do not.

To resume in a different context, the process should have established the current execution context by calling the `sigsetjmp()` function. Instead of exiting normally from the signal handler, the signal handler calls the `siglongjmp()` function. The process execution context reverts to the state saved by the `sigsetjmp()` function call. If the process signal mask was also saved in the call to the `sigsetjmp()` function, the mask is also restored.

The jump functions allow a process to change the flow of control and return to a known process execution context. Changing the flow of control using the jump functions can allow a process to continue executing when it might otherwise terminate abnormally.

Considerations for Using the Jump Functions

The jump functions can be valuable tools in your application program. However, if the program changes the values of variables that are local to the procedure containing the call to `setjmp()`, the program cannot depend upon the values of the local variables being preserved.

If the program must depend on the preservation of a local variable after calls to the jump functions, you must declare the local variable as type `volatile`.

HP Extensions

Table 25-4 shows the HP signals extensions that native processes can use to handle signals.

Table 25-4. HP Signals Extensions to the POSIX.1 Standard

C Function	pTAL Procedure	Description
`SETLOOPTIMER()`	SETLOOPTIMER	Sets the process-loop timer value of the calling process. If the timer expires, a `SIGTIMEOUT` signal is generated.
`SIGACTION_INIT_()`	SIGACTION_INIT_	Establishes the initial state of signal handling for the calling process. This procedure is a replacement for the `ARMTRAP` procedure.
`SIGACTION_RESTORE_()`	SIGACTION_RESTORE_	Restores the signal-handling state stored by a call to the `SIGACTION_SUPPLANT_()` function.
`SIGACTION_SUPPLANT_()`	SIGACTION_SUPPLANT_	Saves the current signal-handling state and allows a subsystem to take over signal handling temporarily.
`SIGJMP_MASKSET_()`	SIGJMP_MASKSET_	Saves the process signal mask in a jump buffer that has already been initialized by the `sigsetjmp()` function or SIGSETJMP_ procedure.

Using HP Extensions

The HP signals extensions are provided as migration and convenience tools that allow native processes to catch signals corresponding to trap conditions in TNS processes. The signals extensions provide shortcuts to the same basic functions as provided by the standard signal interfaces.

If you are concerned about conforming to the POSIX.1 standard and application portability, you should use the standard signals functions. If you are mainly interested in getting the performance gains of converting from TNS to native processes and want to focus on handling those signals known as trap conditions in TNS processes, use the signals extensions.

SIGACTION_INIT_()

The `SIGACTION_INIT_()` function establishes the initial state of signal handling for a process. This function is designed to be called once and sets the process signal mask to unblock all signals. Any signals that are pending when `SIGACTION_INIT_()` is called are discarded.

The `SIGACTION_INIT_()` function installs a signal handler for most signals, including those corresponding to trap conditions. If a nondeferrable signal is delivered and it is blocked, the process terminates abnormally.

Signals can be nested, which means that if a signal is delivered while a signal handler is executing for a different signal, the current signal handler is interrupted and the signal handler for the most-recently received signal is executed. (Traps cannot be nested in TNS processes. If a process receives a trap while a trap handler is executing, the process terminates abnormally.)

If a signal is nondeferrable and the signal handler returns normally, the process terminates abnormally. In this case, the process should exit using the `siglongjmp()` function.

SIGJMP_MASKSET_()

The `SIGJMP_MASKSET_()` function saves the process signal mask in a jump buffer that has already been initialized by the `sigsetjmp()` function. You can save some overhead processing by not saving the process signal mask in each `sigsetjmp()` call and instead calling `SIGJMP_MASKSET_()` before calling `siglongjmp()`.

SIGACTION_SUPPLANT() and SIGACTION_RESTORE()

The `SIGACTION_SUPPLANT_()` function saves the current signal-handling state and allows a subsystem, such as a shared run-time library, to take over signal handling temporarily. Before exiting the signal handler, the subsystem calls the `SIGACTION_RESTORE_()` function to restore the signal-handling state stored by a call to the `SIGACTION_SUPPLANT_()` function.

`SIGACTION_SUPPLANT_()` sets the process signal mask so that all signals that can be blocked are blocked from delivery. Signals that can be deferred, which are those sent to a process by itself and those generated by timers, remain pending until the process exits the subsystem. Nondeferrable signals, which are generated by the system to indicate a run-time error in the process, are delivered to the signal handler.

Interoperability Considerations

The following are some considerations you should be aware of when writing code to handle traps in TNS processes and signals in native processes.

- A Guardian process, whether it is a TNS or native process, cannot contain a mixture of TNS and native procedures and functions. For example, a Guardian native process cannot call the ARMTRAP procedure, and a Guardian TNS process cannot call the `sigaction()` function.

Examples

The first two examples show signal handlers you can use to replace trap handlers that use the ARMTRAP procedure. The programs perform equivalent function; one is in pTAL and one is in C.

The third example shows a signal handler that uses all the HP signal extension functions and history procedures. This example is in C.

Using SIGACTION_INIT_: an Example in pTAL

```
--You can use this program as a replacement for a trap
--handler that calls the ARMTRAP procedure.  This program
--shows how to do the following:
--1. Install a signal handler using the SIGACTION_INIT_
--procedure.
--2. Save the process execution context (including the
--process signal mask) and establish the location to return
--(jump) to from the handler using the SIGSETJMP_ procedure.
--3. Restore the process execution context and perform the
--nonlocal goto (jump) using the SIGLONGJMP_ procedure.

?EXPORT_GLOBALS

?NOLIST, SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL
?LIST
?NOLIST, SOURCE $SYSTEM.SYSTEM.HTDMSIG
?LIST
?NOLIST, SOURCE $SYSTEM.SYSTEM.HSETJMP
?LIST
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0( INITIALIZER,
?        DNUMOUT, PROCESS_GETINFO_, FILE_OPEN_, WRITE )
?LIST
```

```
SIGJMP_BUF_DEF( ENV );
INT TERMNUM;

PROC MYHANDLER (SIGNO, SIG_INFO, SIG_CONTEXTP);
INT(32)   SIGNO;    !signal number delivered to this handler
SIGINFO_T SIG_INFO; !NULL
INT .EXT  SIG_CONTEXTP( UCONTEXT_T );!pointer to saved
                                     !process execution
                                     !context

BEGIN
   STRING BUF [0:40];
   STRING .SBUF;

   BUF  ':=' "Signal " -> @SBUF;
   @SBUF := @SBUF [ DNUMOUT( SBUF, SIGNO, 10 ) ];
   SBUF ':=' " occurred" -> @SBUF;
   CALL WRITE( TERMNUM, BUF, (@SBUF '-' @BUF) '<<' 1);

   --Signal-handling code goes here.  For example, a
   --combination of calls to HIST_* procedures and the
   --information provided in SIG_CONTEXTP can be used to
   --format and display the execution context of the process
   --when the signal occurred.

   --SIGLONGJMP_ restores the process execution context
   --saved by SIGSETJMP_, which is called from the MAIN
   --procedure, and jumps to the location of SIGSETJMP_ with
   --a return value of 1.

   SIGLONGJMP_( ENV, 1D );
END;

PROC M MAIN;

BEGIN
   LITERAL MAXLEN = ZSYS^VAL^LEN^FILENAME;
   INT I := 0;
   INT LEN;
   INT TERMNAME[ 0:MAXLEN-1 ];

   ! Read startup message

   CALL INITIALIZER;

   ! Install the signal handler

   IF SIGACTION_INIT_( @MYHANDLER) <> 0D THEN
      ;   ! Code to handle errors returned by SIGACTION_INIT_

   ! Open home terminal

   CALL PROCESS_GETINFO_( !proc^handle! ,
                          !proc^descriptor:maxlen! ,
                          !proc^descriptor^length! ,
                          !priority!          ,
                          !moms^processhandle! ,
                          TERMNAME:MAXLEN,
                          LEN );
   CALL FILE_OPEN_( TERMNAME:LEN, TERMNUM );
```

```
  --SIGSETJMP_ returns 0 (zero) if called directly and
  --returns a nonzero value if returning from a call
  --to SIGLONGJMP_.

   IF SIGSETJMP_( ENV, 1D ) = 0D THEN
      BEGIN

      --Code that could generate a signal that is caught by
      --MYHANDLER.

      i := 3/i; ! SIGFPE generated here is caught by
                ! MYHANDLER
      END

   ELSE
      BEGIN

      --This is the return location for SIGLONGJMP_, which is
      --called from MYHANDLER after dealing with the signal.

      END;
END;
```

Using SIGACTION_INIT(): an Example in C

```
/* You can use this program as a replacement for a trap
handler that calls the ARMTRAP procedure.  This program shows
how to do the following:
1. Install a signal handler using the SIGACTION_INIT_()
function.
2. Save the process execution context (including the
process signal mask) and establish the location to return
(jump) to from the handler using the sigsetjmp() function.
3. Restore the process execution context and perform the
nonlocal goto (jump) using the siglongjmp() function. */

#include <tdmsig.h>
#include <setjmp.h>

sigjmp_buf env;

void myHandler (signo, sig_info, sig_contextP)
int       signo; /* signal number delivered to this handler*/
siginfo_t *sig_info;    /* NULL */
void      *sig_contextP;/* pointer to saved process */
                        /* execution context */

{
   printf ("Signal %d occurred\n", signo);

   /* Signal-handling code goes here.  For example, a
   combination of calls to HIST_* functions and the
   information provided in sig_contextP can be used to
   format and display the execution context of the
   process when the signal occurred. */

   /* siglongjmp() restores the process execution
   context saved by sigsetjmp(), which is called from
   the main() function, and jumps to the location of
   sigsetjmp() with a return value of 1. */

   siglongjmp (env, 1);
}

main ()
{
   int i = 0;
   if (SIGACTION_INIT_ (myHandler)) /* install the signal */
                                    /* handler */
      ; /* Code to handle errors returned by */
        /* SIGACTION_INIT_() */

   /* sigsetjmp() returns 0 (zero) if called directly and
   returns a nonzero value if returning from a call to
   siglongjmp(). */

   if (! sigsetjmp (env, 1)) {

      /* Code that could generate a signal that is caught
      by myHandler(). */

      i = 3/i; /* SIGFPE generated here is caught by */
               /* myHandler() */

   } else {

      /* This is the return location for siglongjmp(),
      which is called from myHandler() after dealing with
      the signal. */

   }
}
```

Using All the HP Signals Extensions

```
/* This program shows how to use the HP signal extension
   functions, jump functions, and HIST_* functions.
   This program does the following:

   1. Installs a general-purpose signal handler called
   globalHandler() using the SIGACTION_INIT_() function.
   2. Saves the process execution context (including the process
   signal mask) and establishes the location to return (jump)
   to from globalHandler() using the sigsetjmp() function.
   3. Restores the process execution context and performs the
   nonlocal goto (jump) using the siglongjmp() function.
   4. Installs a local signal handler called localHandler(),
   which takes over signal handling from globalHandler() for
   system-generated nondeferrable signals (such as SIGFPE) using
   the SIGACTION_SUPPLANT_() function. The signal-handling
   state established by globalHandler() is saved.
   5. Saves the process execution context and establishes the
   location to return (jump) to from localHandler() using the
   setjmp() function.
   6. Formats and displays the process execution context when
   the signal occurred using the HIST_* functions and the
   information provided in sig_contextP.
   7. Unblocks all signals by clearing the process signal mask
   in the jump buffer using the SIGJMP_MASKSET_() function.
   8. Restores the process execution context (including the
   process signal mask) and jumps to the location established
   by the call to setjmp(), using the siglongjmp() function.
   9. Restores the signal-handling state in globalHandler(),
   which was installed by SIGACTION_INIT_() and saved by
   SIGACTION_SUPPLANT_(), using the SIGACTION_RESTORE_()
   function. */

#include <tdmsig.h> nolist
#include <setjmp.h> nolist
#include <histry.h> nolist
#include <stdio.h>  nolist
#include <stdlib.h> nolist

jmp_buf jmpEnv;
sigjmp_buf sigJmpEnv;

void localHandler    /* local signal handler for "worker" */
         ( int signo             /* signal number */
         , siginfo_t * siginfo   /* NULL */
         , void * sig_contextP   /* pointer to saved process */
         )
{
   NSK_histWorkspace hws;
   int error;
   char buf [80];

   printf ("localHandler: Signal %d occurred\n", signo);

   /* Use HIST_FORMAT_() to produce an ASCII text
      representation of the process execution context for the
      process indicated by HIST_INIT() or HIST_GETPRIOR_(). */

   error = HIST_INIT_ (&hws, HistVersion1, HO_Init_uContext,
                       sig_contextP);
   if (error == HIST_OK)
      do {
         int len;
         while ((len = HIST_FORMAT_ (&hws, buf, 79)) > 0) {
            buf[ len ] = 0;
            printf ("%s\n", buf);
            }
        } while ((error = HIST_GETPRIOR_ (&hws)) == HIST_OK);

    else
        printf ("HIST_INIT_ error: %d. Unable to trace\n",
                error);

    /* if error != HIST_DONE ... */
    SIGJMP_MASKSET_ (jmpEnv, (sigset_t *) 0); /* unblock the
signals */

    /* Restore the process execution context (including the
        process signal mask) and jump to the location of the
        setjmp() call in worker(). */

    siglongjmp (jmpEnv, 1);

}

int divider ( int i, int j ) { /* divide i by j */
    return ( i/j ); /* When j is zero, and the program was
compiled with the
                      overflow_traps directive, generates a
SIGFPE signal */
}

void doMoreProcessing ()
{

    /* Generate a SIGFPE signal caught by localHandler(). */
    divider( 3, 0 );
    /* We don't expect divider() to return. */
    printf( "doMoreProcessing: after (unexpected) return from
divider()\n" );
}

int worker ()
{

    int result;

    sig_save_template sigSaveBuf;
    if (SIGACTION_SUPPLANT_ (localHandler, &sigSaveBuf,
                             sigsave_len))
        return 1;   /* returns 1 for failure */
    /* Now we are in the domain of localHandler().*/

    if (! setjmp (jmpEnv)) {
        /* Normal path: nondeferrable signals are handled by
           localHandler(). */
        doMoreProcessing ();
```

```
      result = 0; /* 0 means success */
   } else {

      /* Error path: entered via siglongjmp() called in
         localHandler(). */
      result = 1;

   }

   if (SIGACTION_RESTORE_ (&sigSaveBuf))
      exit (2);
   return result;
}

void globalHandler
      ( int signo            /* signal number */
      , siginfo_t *sig_info  /* NULL */
      , void *sig_context    /* pointer to saved process */
      )
{

   printf ("globalHandler: Signal %d occurred\n", signo);
   /* A combination of HIST_* functions and the information
      provided in sig_context can be used to format and display
      the process execution context when the signal occurred. */
   /* Restore the process execution context (including the
      process signal mask) saved by sigsetjmp() called from
      main() and jump to the location of the sigsetjmp() call
      with a return value of 1. */

   siglongjmp (sigJmpEnv, 1);

}

main ()

{

   printf( "main: start of execution\n" );
   if (SIGACTION_INIT_ (globalHandler)) /* install the signal
handler */

   {

      printf( "main: SIGACTION_INIT failed\n" );
      exit (1);

   }

   if (! sigsetjmp (sigJmpEnv, 1)) {
      printf( "main: after sigsetjmp returned 0\n" );
      /* Code that could generate a signal that is caught
         by globalHandler(). */

      if (worker ()) {/* worker() establishes its own */
         /* signal handling domain with */
         /* localHandler(). */
         /* Code to deal with errors returned from worker(). */

      }

      /* Back once again in the domain of globalHandler()
         installed by SIGACTION_INIT_(). */
      /* SIGFPE generated here is caught by globalHandler() */
      divider ( 5, 0 );
      /* If traps are enabled, we shouldn't get here. */
      printf( "main: second call to divider() failed to generate
                signal\n" );

   } else {

      /* The siglongjmp() call in globalHandler() gets used
      here. */
      printf( "main: after sigsetjmp returned nonzero\n" );
   }

}
```

26 Synchronizing Processes

One or more processes executing concurrently on a HP system might need to share a particular resource. Most commonly, the shared resource is an area of memory, such as an extended data segment. This sharing can result in conflicts and possible errors. Consider, for example, two processes, A and B, running concurrently in the same CPU, both of which are attempting to increment a variable in a shared memory area. Process A fetches the variable, adds 1 to it, and returns it to memory. Process B then fetches the variable, adds 1 to it, returns it to memory, and proceeds under the assumption that the current value of the variable is 2 greater than the original value. But suppose the processes are synchronized such that process B fetches the variable just before A returns it to memory. The variable would not have the correct value.

One solution would be to simply run the processes sequentially to ensure that there is no overlap in their execution and, consequently, no conflicts in accessing the shared resource. However, this solution would result in very inefficient processing that would not take advantage of the HP system's ability to run multiple processes concurrently. Clearly, a solution is needed that allows concurrent processes to access a shared resource without conflicts while continuing to execute in parallel.

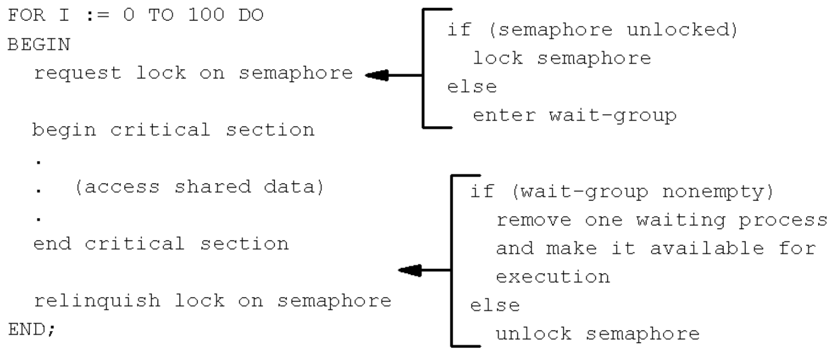
Binary semaphores provide such a solution. Binary semaphores are a tool to synchronize processes. They provide a way for a process to hold exclusive access to a resource while accessing the resource. Using binary semaphores, you can synchronize processes so that only one process at a time can use a shared resource. While a process is accessing the resource, other processes can execute concurrently until they need to use the resource. They then enter a wait state until the resource becomes available. When the original process is through with the resource, it releases its "hold" on the resource, and another process in the waiting group acquires exclusive access to the resource and resumes execution.

Note. Binary semaphores enable you to synchronize processes running in the same CPU; to synchronize processes in different CPUs, you must use other techniques. However, when the issue is serializing access to variables, as in the above example, the variables must be in shared memory, which implies that the sharers are on the same CPU.

How Binary Semaphores Work

Using binary semaphores, a programmer can maximize parallelism (the degree to which processes are able to execute concurrently) among concurrent processes while avoiding conflicts over shared resources. A binary semaphore consists of a global entity called a **lock** and an associated group of waiting processes called a **wait group**. Figure 26-1 illustrates the binary semaphore concept.

Figure 26-1. Binary Semaphore



```
FOR I := 0 TO 100 DO
BEGIN
  request lock on semaphore

  begin critical section
  .
  .   (access shared data)
  .
  end critical section

  relinquish lock on semaphore
END;
```

```
if (semaphore unlocked)
  lock semaphore
else
  enter wait-group
```

```
if (wait-group nonempty)
  remove one waiting process
  and make it available for
  execution
else
  unlock semaphore
```

VST130.VSD

In the above example, the executing process follows these steps:

1. When the executing process reaches a **critical section** (a sequence of code that accesses a shared resource), it requests a lock on the semaphore.
2. If the semaphore is unlocked, the process locks it and executes the critical code. If the semaphore is locked by another process, the process requesting the lock is placed in the wait group.
3. When the process holding the lock finishes executing its critical section, it relinquishes the lock on the semaphore. The effect of relinquishing the lock is to check the wait group and do one of the following:
    - If processes are waiting for the lock, one is removed from the group and made available for execution; that process takes ownership of the lock.
    - If no processes are waiting, the semaphore is unlocked and the next process to enter a critical section can lock the semaphore and access the shared resource.

Table 26-1 lists four processes, indicated by A, B, C, and D, using binary semaphores to synchronize their access to a shared memory area. The table shows the state of each process at arbitrary points in time represented by t0 through t5.

Table 26-1. Process Synchronization

	Not Executing	Executing Non-critical Section	Executing Critical Section	Waiting
t0	A,B,C,D			
t1	A	B	D	C
t2			D	A,B,C
t3		C,D	A	B
t4	D	A,C	B	
t5	A,B,C,D			

At t0, the four processes have not been started.

At t1, process D is accessing the shared area, C has reached a critical section and has entered the wait group, B has been started and is executing a noncritical section, and A is still waiting to be started.

At t2, D is still holding the lock, and A, B, and C are waiting for the lock.

At t3, C and D are executing noncritical sections, A now owns the lock and has exclusive access to the shared area, and B is still waiting.

At t4, D has finished executing, A and C are executing noncritical sections, and B has access to the shared area.

Finally, at t5, all four processes have finished.

Summary of Guardian Binary Semaphore Procedures

HP provides Guardian procedure calls to implement the binary semaphore capability. The procedures are callable from programs written in Transaction Application Language (TAL) and C/C++. The binary semaphore procedure calls are summarized in Table 26-1. Refer to the *Guardian Procedure Calls Reference Manual* for detailed descriptions of these procedure calls, including input and output parameters and error status values returned by the calls.

Table 26-2. Binary Semaphore Procedure Calls

Procedure	Description
BINSEM_CREATE_	Creates, opens, and locks a binary semaphore. Generally called by the program that creates the shared resource.
BINSEM_OPEN_	Opens a binary semaphore. All processes that will use the binary semaphore must first open it.
BINSEM_LOCK_	Locks a binary semaphore. Called by a process immediately before entering a critical section. Only one process at a time can lock a binary semaphore. This enables a program to have exclusive access to a shared resource. If a process tries to lock a semaphore that is already locked, the process enters a wait state.
BINSEM_UNLOCK_	Relinquishes a lock on a binary semaphore. Called by a process immediately upon leaving a critical section. If there are processes in the wait group, one of those processes is given ownership of the lock.
BINSEM_CLOSE_	Closes a binary semaphore. Called by a process when it is finished accessing the binary semaphore. When the last process that has a semaphore open closes the semaphore, that semaphore ceases to exist.
BINSEM_FORCELOCK_	Forces a lock on a binary semaphore. Used to take a lock away from the process that has the lock and has entered an unresponsive state (for example, an infinite loop).
BINSEM_GETSTATS_*	Returns counter statistics for one or more binary semaphores for a specified process. Additionally, BINSEM_GETSTATS_ can clear counters and reset the maximum number of contenders.
BINSEM_STAT_VERSION_*	Accepts a version number defined in `kbinsem(.h)` for the BINSEM_GETSTATS_ procedure and checks whether or not it matches the implementation version of BINSEM_GETSTATS_.

* Available only for NonStop systems running J06.14, H06.25, and subsequent RVUs.

Using the Binary Semaphore Procedure Calls

This subsection describes the steps involved in using binary semaphores. Following is a summary of these steps:

1. First, the binary semaphore must be created (call BINSEM_CREATE_).
2. After the binary semaphore is created, all processes that will access the binary semaphore must open it (call BINSEM_OPEN_). The process that creates the binary semaphore does not need to call BINSEM_OPEN_, because BINSEM_CREATE_ also opens the binary semaphore.
3. Before executing a critical section of code (that is, before accessing a shared resource), a process locks the binary semaphore (call BINSEM_LOCK_). The process that creates the binary semaphore does not need to call BINSEM_LOCK_, the first time it attempts to lock the semaphore because BINSEM_CREATE_ also locks the binary semaphore.
4. After finishing the critical section, a process unlocks the binary semaphore so that another process can lock it and safely access the resource (call BINSEM_UNLOCK_).
5. Once a process is finished using a binary semaphore, it should close the binary semaphore to free any system resources used by the binary semaphore (call BINSEM_CLOSE_).

An additional procedure, BINSEM_FORCELOCK_, is provided to enable a process to take a lock away from the process currently holding the lock. This call should be used only if absolutely necessary.

Creating a Binary Semaphore

The first step in synchronizing a group of processes is to create a binary semaphore. In general, the process that creates the resource to be shared also creates the binary semaphore, although this is not a requirement.

To create a binary semaphore, call the BINSEM_CREATE_ procedure. This procedure creates, opens, and locks a binary semaphore, and it returns a semaphore ID that is used to refer to the semaphore in subsequent calls. The process that creates the binary semaphore must make this semaphore ID, along with its own process handle, available to other processes that will access the semaphore. This can be done by placing the values in a shared data area, as illustrated in the example at the end of this section.

In the BINSEM_CREATE_ call, you must also specify a security level for the semaphore. The security level determines which other processes are permitted to open the semaphore. For example:

```
SECURITY := 0;
ERROR := BINSEM_CREATE (SEMID, SECURITY);
```

In this example, a binary semaphore is created with security level 0, which permits all processes in the same CPU to access the semaphore.

Opening a Binary Semaphore

All processes that will access a binary semaphore must first open the binary semaphore. The BINSEM_CREATE_ call creates and opens a binary semaphore; all other processes besides the process that calls BINSEM_CREATE_ must call BINSEM_OPEN_. A process generally calls BINSEM_OPEN_ at the beginning of its execution. To synchronize access to several shared resources, a process can have multiple binary semaphores open at the same time. (The current binary semaphore limits are 8,000 per process and 64,000 per processor.)

When calling BINSEM_OPEN, you must specify the following input parameters:

- The process handle of either the process that created the semaphore or a process that previously opened the semaphore.
- The ID of the semaphore being opened (the ID returned by the above process).

The process that created or previously opened the semaphore must communicate these values to all other processes that will open the semaphore.

BINSEM_OPEN_ returns an ID value that identifies the binary semaphore locally in this process; it is different from the semaphore ID passed as input to BINSEM_OPEN_. The local ID is used to designate the binary semaphore in subsequent calls within the same process; it can also be conveyed to other processes for use as input to BINSEM_OPEN_ or BINSEM_GETSTATS_, along with the process handle of the process to which it belongs.

Locking a Binary Semaphore

Locking a binary semaphore enables a process to exercise exclusive access to a shared resource; only one process at a time can lock a binary semaphore. To lock a binary semaphore, call the BINSEM_LOCK_ procedure. A process typically calls BINSEM_LOCK_ just before accessing the shared resource.

If the binary semaphore is unlocked when BINSEM_LOCK_ is called, the calling process is granted a lock on the semaphore and can access the shared resource safely.

If another process has the binary semaphore locked when BINSEM_LOCK_ is called, execution of the calling process is suspended and the process is placed in a wait group. The process remains in this state until either:

- The process currently holding the lock unlocks the binary semaphore, and the suspended process is selected from the wait group and granted the lock.

  or

- The value specified for the *`timeout`* parameter in the BINSEM_LOCK_ call is reached. The *`timeout`* parameter provides a way to ensure that a process does not wait indefinitely for a lock. If the timeout value is reached before the process is selected to receive the lock, the process is awakened and EAGAIN is returned from BINSEM_LOCK.

When the process currently holding the lock unlocks the binary semaphore, the wait group is checked for waiting processes. If the wait group contains waiting processes, a process is selected from the wait group to receive ownership of the lock and resume execution at the point of suspension. The selected process is then free to access the shared resource.

Note. The order in which processes are selected from the wait group to receive ownership of a lock is indeterminate; processes are not necessarily selected in the order in which they entered the group nor are they selected in order of priority.

If a process holding a lock on a semaphore terminates without unlocking the semaphore, that semaphore is said to be forsaken. The next process in the wait group or, if the wait group is empty, the next process to call BINSEM_LOCK_, gets the lock but is informed by the system that the semaphore has not been unlocked by the original process. Because the shared resource may be in an inconsistent state, the process requesting the lock should perform an application-dependent recovery before using the resource.

In the following example, a lock is requested and given a timeout value of 500 seconds:

```
TIMEOUT := 50000;   !timeout is expressed in units of 10
                     milliseconds
ERROR := BINSEM_LOCK_ (SEMID, TIMEOUT);
```

Unlocking a Binary Semaphore

When a process has finished executing a critical section and is ready to relinquish a lock on a binary semaphore, it should unlock the binary semaphore. To unlock a binary semaphore, call BINSEM_UNLOCK_.

The BINSEM_UNLOCK_ procedure unlocks a binary semaphore and makes it available to other processes. If other processes are in the wait group when BINSEM_UNLOCK_ is called, a waiting process is selected to receive the lock and the binary semaphore is immediately locked. If no processes are in the wait group, the binary semaphore is available to the next process that requests the lock.

BINSEM_UNLOCK_ cannot unlock a binary semaphore that is currently locked by another process; to do so, use the BINSEM_FORCELOCK_ procedure to force a lock on the binary semaphore, then call BINSEM_UNLOCK_.

Testing Ownership of a Binary Semaphore

Software should be designed so that a process is always aware of a locked binary semaphore, but sometimes a common-code sequence can be invoked either with or without the semaphore. Such code can call the BINSEM_ISMINE_ procedure to determine whether or not a binary semaphore is locked by this process.

BINSEM_ISMINE_ returns a nonzero value if the SEMID is valid and the calling process currently has that binary semaphore locked.

Forcing a Lock on a Binary Semaphore

Under certain conditions you may want to force a lock on a binary semaphore that is currently locked by another process. For example, a process holding a lock may have entered an infinite loop or some other unresponsive state. To force a lock on a binary semaphore, call the BINSEM_FORCELOCK_ procedure.

The BINSEM_FORCELOCK_ procedure enables a process to take a lock away from another process. However, the original process continues to run as though it still has the lock, so to avoid possible conflicts, you should take steps to terminate that process.

In order to force a lock on a binary semaphore, a process must have permission to access the binary semaphore, as specified in the BINSEM_CREATE_ call that created the binary semaphore.

Note. The BINSEM_FORCELOCK_ call should be used only in critical situations, because it circumvents the protection that binary semaphores are designed to provide.

Closing a Binary Semaphore

Once a process is finished executing critical sections of code, it should close any binary semaphores that it has open. To close a binary semaphore, call the BINSEM_CLOSE_ procedure.

A process typically calls BINSEM_CLOSE_ at the end of its execution. After closing a binary semaphore, the process no longer has knowledge of it and can no longer request a lock on it. Once all processes that have opened a binary semaphore close it, any resources used by the semaphore are returned to the system.

When a process terminates, any binary semaphores it has open are automatically closed. If any are locked by the process at the time of termination, they become forsaken, as described earlier under Locking a Binary Semaphore.

Binary Semaphore Interface Declarations

As of the J06.14 and H06.25 RVUs, the entire binary semaphore interface is defined in two new header files: KBINSEM for TAL/epTAL and KBINSEMH (also referred to as `kbinsem.h`) for C/C++. These headers include literals for options and result codes, the output structure for BINSEM_GETSTATS_, and the version literals for BINSEM_STAT_VERSION_. In all supported releases, six of the BINSEM_... procedures continue to be declared in EXTDECS* and CEXTDECS. The three newer procedures, BINSEM_ISMINE_, BINSEM_GETSTATS_, and BINEM_STATS_VERSION_, are declared only in KBINSEM and `kbinsem.h`.

The `kbinsem.h` file also contains type definitions and macros to manage the process handle parameters required by two of the procedures: BINSEM_GETSTATS_ and

BINSEM_OPEN_. These handles can be represented either as a structure or as an `array[10]` of `short int`. Guardian procedures declared in CEXTDECS use the array form. Some interfaces use a NSK_PHandle structure type, defined in the `kphandl.h` header file. The `kbinsem.h` file can work either way.

By default, `kbinsem.h` uses the structure. Thus a program could contain:

```
#include <kbinsem.h>
NSK_PHandle myPhandle;
...
ret = PROCESSHANDLE_GETMINE((short*)&myPhandle);
...
ret = BINSEM_GETSTATS_(&myPhandle, ...);
```

or

```
#include <kbinsem.h>
short myPhandle[10];
...
ret = PROCESSHANDLE_GETMINE(myPhandle);
...
ret = BINSEM_GETSTATS_((NSK_Phandle*)myPhandle, ...);
```

Alternatively, to use the array form, the program could contain:

```
#include <CEXTDECS>    /* or #define PHANDLE_IS_STRUCT 0 */
#include <kbinsem.h>
short myPhandle[10];
...
ret = PROCESSHANDLE_GETMINE(myPhandle);
...
ret = BINSEM_GETSTATS_(myPhandle, ...);
```

By using macros defined in `kbinsem.h`, source code can be made agnostic with respect to the process handle type:

```
binSemPhanDef(myPhandle);
...
ret = PROCESSHANDLE_GETMINE(binSemPHanShortPtr myPhandle);
...
ret = BINSEM_GETSTATS_(binSemPhanPtr myPhandle, ...);
```

The declarations in `kbinsem.h` and those in CEXTDECS are compatible under either of the following circumstances:

- The CEXTDECS file is not included whole, and the BINSEM_OPEN_ section is not included. By default, `kbinsem.h` defines PHANDLE_IS_STRUCT as 1 and uses the structure pointer type.
- The CEXTDECS file is included whole, before the `kbinsem.h` file. In this case, `kbinsem.h` defines PHANDLE_IS_STRUCT as 0 and uses the array representation.

For details, see the `kbinsem.h` header file.

Binary Semaphore Example

Following is a simple TAL example illustrating the use of the binary semaphore procedure calls. This example can be used as a template for creating other programs that use binary semaphores. The example assumes that two processes share a segment containing a shared structure. One process executes the PRIMARY procedure and the other process executes the SECONDARY procedure. The address of the shared structure is passed to these procedures. The program consists of the following components:

- The shared memory segment. This segment contains both the shared structure and the variables comprising the resource to which access is controlled.
- The shared structure. This structure is used to pass the process handle of the main process and the binary semaphore ID from the primary process to the secondary process.
- The external declarations that provide access to the binary semaphore procedures and other Guardian procedures used by the program.
- The USERESOURCE procedure. This procedure accesses the shared resource. It is called by the main process and the secondary process.
- The PRIMARY procedure. This procedure is executed by the main process. It creates the binary semaphore, creates the secondary process, and uses the shared resource.
- The SECONDARY procedure. This procedure accesses the shared resource.

Shared Structure

The shared structure is used to pass the main process handle and the semaphore ID from the main process, which creates the binary semaphore, to the secondary process, which opens the binary semaphore. The structure is as follows:

```
STRUCT SHARED_TEMPLATE (*);
BEGIN
   INT MAINPROCESSHANDLE[0:9];
   INT(32) MAINSEMID;
END;
```

External Declarations

The following SOURCE directives provide access to the binary semaphore procedures and other Guardian procedures used by the program:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(BINSEM_CREATE_, BINSEM_OPEN_)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(BINSEM_CLOSE_, BINSEM_LOCK_)
?SOURCE.$SYSTEM.SYSTEM.EXTDECS0(BINSEM_UNLOCK_, DEBUG)
?SOURCE.$SYSTEM.SYSTEM.EXTDECS0(PROCESS_GETINFO_)
?SOURCE.$SYSTEM.SYSTEM.EXTDECS0(PROCESSHANDLE_NULLIT_)
```

Procedure USERESOURCE

Procedure USERESOURCE locks the binary semaphore, uses the shared resource, and unlocks the binary semaphore. The procedure is called by the primary process and the secondary process. Procedure USERESOURCE is as follows:

```
PROC USERESOURCE (SEMID);
INT(32) SEMID;
BEGIN
INT ERROR;

ERROR := BINSEM_LOCK_ (SEMID, -1D);
IF (ERROR <> 0) THEN CALL DEBUG;

... Use the shared resource...

ERROR := BINSEM_UNLOCK_ (SEMID);
IF (ERROR <> 0) THEN CALL DEBUG;

END;
```

Procedure PRIMARY

Procedure PRIMARY is executed by the main process. That process allocates a memory segment to contain both the shared structure and the shared resource. It creates a pointer to the structure and passes that pointer to PRIMARY. This procedure creates the binary semaphore and makes the main process handle and semaphore ID available to other processes by placing them in the shared structure. The PRIMARY procedure then creates the secondary process, uses the shared resource, and closes the binary semaphore. The PRIMARY procedure is as follows:

```
PROC PRIMARY (SHARED);
INT .EXT SHARED (SHARED_TEMPLATE);
BEGIN
INT ERROR;

ERROR := PROCESSHANDLE_NULLIT_ (SHARED.MAINPROCESSHANDLE);
IF (ERROR <> 0) THEN CALL DEBUG;
ERROR := PROCESS_GETINFO_ (SHARED.MAINPROCESSHANDLE);
IF (ERROR <> 0) THEN CALL DEBUG;
ERROR := BINSEM_CREATE_ (SHARED.MAINSEMID, 2);
IF (ERROR <> 0) THEN CALL DEBUG;


...Create secondary process ...

CALL USERESOURCE (SHARED.MAINSEMID);

ERROR := BINSEM_CLOSE_ (SHARED.MAINSEMID);
IF (ERROR <> 0) THEN CALL DEBUG;

END
```

Procedure SECONDARY

Procedure SECONDARY is executed by the secondary process. That process shares the segment created by the primary process, and passes to SECONDARY a pointer to the shared structure. It opens the binary semaphore, locks the binary semaphore, uses the shared resource, unlocks the binary semaphore, and closes the binary semaphore. The main process handle and the semaphore ID are picked up from the shared structure. Procedure SECONDARY is as follows:

```
PROC SECONDARY (SHARED);
INT .EXT SHARED (SHARED_TEMPLATE);
BEGIN
INT ERROR;
INT(32) SECONDARYSEMID;

ERROR := BINSEM_OPEN_ (SECONDARYSEMID,
                       SHARED.MAINPROCESSHANDLE,
                       SHARED.MAINSEMID);
IF (ERROR <> 0) THEN CALL DEBUG;

CALL USERESOURCE (SECONDARYSEMID);

ERROR := BINSEM_CLOSE_ (SECONDARYSEMID);
IF (ERROR <> 0) THEN CALL DEBUG;

END;
```

BINSEM_GETSTATS_ and BINSEM_STAT_VERSION_ Example

Beginning with the J06.14 and H06.15 RVUs, the BINSEM_GETSTATS_ and BINSEM_STAT_VERSION_ procedures are available. See the *Guardian Procedure Calls Reference Manual* for details of these procedures.

The following `binsemc` program is an example of using the BINSEM_GETSTATS_ and BINSEM_STAT_VERSION_ procedures. It illustrates how a program can display statistics about its own use of binary semaphores. Alternatively, the user could invoke the SEMSTAT utility documented in the *TACL Reference Manual*.

```
#include <kbinsem.h> _nolist
#include <cextdecs(PROCESSHANDLE_GETMINE_)>  _nolist
#include <stdio.h>   _nolist

binSemID_t semID;
short err;
unsigned int ret;
binSemStats_t stat;
NSK_PHandle pHandle;

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

```
{
    /* Create a BinSem */
    err = BINSEM_CREATE_(&semID, 2);
    if(err !=BINSEM_RET_OK) {
       printf("BINSEM_CREATE_ failed with status %d\n", err);
       return(-1);
    }
    /* Unlock BinSem Created */
    err = BINSEM_UNLOCK_(semID);
    if (err)
        printf("BINSEM_UNLOCK_ failed with %d\n", err);

    /* A real program would have application logic,
       including locking and unlocking one or more binary
       semaphores.
       It might also create or open additional semaphores.
        ...
    */
    err = PROCESSHANDLE_GETMINE_((short*)&pHandle);
    if(err)
       printf("PROCESSHANDLE_GETMINE_ failed with error:%d\n",
                err);

    /* Verify BinSem Stats Version */
    ret = BINSEM_STAT_VERSION_(BINSEM_STAT_VERSION1);
    if(ret < sizeof(binSemStats_t))
       printf("BINSEM_STAT_VERSION_ failed with error:%d\n",
                ret);

    /* Print BinSem Stats */
    puts("Sem ID     Acquisitions    Tot. Cont.   Mult. Cont.  "
           "Cur. Cont.  Max. Cont");

    semID = BINSEM_STAT_INIT_CONTEXT;
    for (;;)
    {
        /* Get BinSem Stats */
        err = BINSEM_GETSTATS_(&pHandle, semID,
                                BINSEM_STAT_OPT_DEFAULT,
                                  &stat, sizeof stat);
        if (err != BINSEM_RET_OK)
            break;

        printf("%6d %17u %13u %13u %11u %10u\n",
                stat.semID, stat.acquisitions,
                stat.contentions,
                stat.multiCont, stat.contenders,
                stat.maxContend);
        semID = stat.semID;
    };
    if(err != BINSEM_RET_EOF)
       printf("BINSEM_GETSTATS_ failed with error %d\n", err);

    if (err = BINSEM_CLOSE_(semID))
        printf("BINSEM_CLOSE_ failed with %8.8x\n", err);
    return 0;
}
```

The `binsemc` example program can be compiled for various data models with the following commands, and then executed:

- `ccomp /in binsemc/ obinsem; symbols, nolist, runnable,& extensions`
- `ccomp /in binsemc/ obinsem; symbols, nolist, runnable,& extensions, systype oss, ILP32`
- `ccomp /in binsemc/ obinsem; symbols, nolist, runnable,& extensions, systype oss, LP64`

27

Fault-Tolerant Programming in C

The term “fault-tolerant” means that a single failure does not cause processing to stop.

At the hardware level, redundant hardware and duplication of paths allow systems to tolerate a single-component failure. In many cases, multiple-component failures can also be tolerated as long as they do not share common paths. Moreover, the redundant paths are not duplicate backups; that is, all available resources are used for processing—none are held in reserve for use as spare backups. The hardware concepts used to achieve this fault tolerance are explained in the *Introduction to Tandem NonStop Systems*.

Software can be written to be fault-tolerant. Many software problems are transient; that is, the problem is caused by an unusual environment state typically resulting from a transient hardware problem, a resource limit exceeded, or a race condition. In such cases, reinitializing the program state to an earlier point and resuming execution often works because the environment is different.

An application does not execute in a fault-tolerant manner automatically; it must be designed and implemented to run as a fault-tolerant program. This section describes the approach to fault-tolerant programming known as **active backup**.

This section includes the following information:

- An overview of the activities an active backup program must perform.
- An overview of the tasks a programmer must complete to create an active backup program.
- A summary of the C language extensions that support active backup programming.
- An explanation of how to organize an active backup program.
- Two examples of active backup programs.

Overview of Active Backup Programming

In active backup programming, processes are executed in pairs: a primary process, which performs the tasks of the underlying application, and a backup process, which is ready to take over execution from the primary process should the primary process or CPU fail. Active backup programs have the following characteristics:

- Active backup uses process pairs to achieve fault tolerance.
- The primary process sends state information to the backup process. State information is information about the run-time environment that is required for the backup to take over for the primary.
- The backup process receives state information from the primary, detects a failed primary process or CPU, and takes over execution.

An active backup program executes as a primary and backup process pair running the same program file. The primary and backup processes communicate through interprocess communication. The primary process sends critical data to the backup process. This critical data serves two purposes: to provide sufficient information to allow the backup to resume application processing and to indicate to the backup where it should logically resume application processing.

The backup process receives messages from two sources. It receives critical information from the primary process, which it must record for future use in the event it must take over processing from the primary. It can also receive messages from the operating system indicating that the primary process or CPU has failed. If the primary process fails, the backup takes over processing at the logical point in the application indicated by the most recent control state information received from the primary, and it continues processing using the most recent file state and application state information.

Summary of Active Backup Processing

When an active backup program is started, it is given a process name. This allows the new process (and later the backup process) to run as a named process pair (use of unnamed process pairs is not discussed in this section). Following are the activities that an active backup program performs:

- A new process determines whether it is executing as the primary process or the backup process.
- If the process is the primary process, it does the following:
    - Opens files required for execution.
    - Creates and starts the backup process (normally in another CPU), and opens it for interprocess communication.
    - Gets open file state information and sends it to the backup process.
    - Begins executing the application statements. At critical points, the primary process updates **state information**; that is, it sends critical file and data information to the backup process.
    - Monitors the backup process. If the backup process or CPU fails, the primary can recognize that and create another backup.
- If the process is the backup process, it enters a message-processing loop. While in this loop, the backup process:
    - Does a backup open of any files required by the application. A backup open is a special open that allows files to be open concurrently by both the primary and backup processes.
    - Monitors the primary process and primary CPU.

      The backup process stays in the message-processing loop until either the primary process fails or the application terminates.

If the primary process or CPU fails, the backup process takes over execution from the failed primary process and becomes the new primary process. First, it creates a new backup process. (If the failure is caused by CPU failure, the new backup is created either immediately in another CPU or when the failing CPU is brought back online.)

The backup process then continues application processing at a point indicated in the state information received from the primary process.

What the Programmer Must Do

When coding a program to run as a process pair, there are several activities you, as the programmer, need to complete. These include planning tasks, which should be completed before coding an active backup program, and programming tasks, which involve the actual coding of an active backup program.

Note that fault-tolerant programs should be designed that way from the outset. Converting existing programs to run in a fault-tolerant manner can be very difficult, depending on the structure of the program.

Planning Tasks

Before coding an application to run as an active backup program, do the following:

- Develop a strategy for updating state information. You will need to include statements in your program for providing the backup process with the information it needs to take over execution if the primary process fails. This state information accomplishes three things:

    - Tells the backup process where to take over execution.
    - Provides critical information about files currently in use by the application.
    - Provides current data values to the backup process.

    You must determine what information to provide and the points in the execution of the application at which the state information will be updated and at which the backup can take over execution. Developing an appropriate strategy is vitally important; errors can result if the backup does not have the correct state information. Guidelines for developing a strategy for updating state information are given later in this section under Updating State Information.

- Define a communications protocol. You need to provide for passing messages between the primary and backup processes. The communications protocol enables the primary to send state information to the backup. It enables the backup to monitor the primary process and CPU and to receive state information from the primary process. The communications protocol should use the same message formats as the operating system uses. HP recommends that you use the Guardian interprocess communication facility. Guidelines are given later in this section under Providing Communication Between the Primary and Backup Processes.

Programming Tasks

Once you have developed a strategy for updating state information and defined a protocol for interprocess communication, you can begin coding your active backup program. Compaq provides extensions to the C language that support active backup programming. These language extensions are summarized later in this section under C Extensions That Support Active Backup Programming. Following is a summary of the programming tasks required or recommended for coding an active backup program. Details for coding an active backup program are given later in this section under Organizing an Active Backup Program.

To code a program to run in a fault-tolerant manner, you must:

- Include statements to determine whether a process is the primary process or the backup process.
- Include statements to start the backup process and open it for interprocess communication.
- Provide a mechanism for sending state information to the backup process.
- Provide a mechanism for the backup process to receive and save state information from the primary process.
- Provide a mechanism for the backup process to receive and process failure messages from the primary process and from the operating system.
- Provide a mechanism for the backup process to take over for the primary process if the primary process or CPU should fail.
- Provide a mechanism for the primary process to detect a failure of the backup process.
- Include statements for detecting and handling duplicate and old requests.
- Include statements for reinitating pending I/Os and pending signal timeouts on takeovers.

You use a combination of programming techniques, Guardian procedure calls, and C-supplied functions to perform these tasks.

C Extensions That Support Active Backup Programming

C provides several functions that you can use to perform various tasks required for active backup programming. This subsection provides a brief overview of these functions. These functions are available for TNS and accelerated programs as well as for TNS/R native programs. For details, refer to the *Guardian TNS C Library Calls Reference Manual* and the *Guardian TNS/R Native C Library Calls Reference Manual*.

The following table summarizes the C functions used for active backup programming:

Function	**Use**
__ns_start_backup	Called by the primary process to create and initialize the backup process.
__ns_fopen_special	Called by the primary process to open a file with a specified sync depth.
__ns_fget_file_open_state	Called by the primary process to obtain open state information for a file.
__ns_backup_fopen	Called by the backup process to back up open files that have already been opened by the primary process.
__ns_fget_file_state	Called by the primary process to obtain file state information.
__ns_fset_file_state	Called by the backup process to update file state information.

Starting the Backup Process

The primary process starts and initializes the backup process by calling the `__ns_start_backup` function. The `__ns_start_backup` function does the following:

- Starts the backup process. You can optionally specify the CPU in which the backup process is to run.
- Optionally returns the process handle of the backup process (assuming the backup process was started successfully).
- Sends to the backup process the same startup, assign, and param messages that the primary process received.
- Gets file open state information for the `stdin`, `stdout`, and `stderr` files that the primary process opened during its initialization, and sends that information to the backup process. The backup process receives the open state information through the $RECEIVE file.

The backup process has the same swap volume as the primary process.

After receiving messages sent by `__ns_start_backup`, the backup process automatically does the following:

- Processes the startup, assign, and param messages in the same manner as the primary process.
- For any of the standard files `stdin`, `stdout`, and `stderr` the primary opened during its initialization, performs backup opens for the same files.

Only the standard files `stdin`, `stdout`, and `stderr` can have backup opens automatically performed. Other files that the primary process may have opened must be explicitly opened in the backup process by a call to `__ns_backup_fopen`.

Note that when the standard files are backup opened, the file states in the backup process are not automatically set to the corresponding file states in the primary process. If operations are performed on a standard file before `__ns_start_backup` is called, it is your responsibility to ensure that the file state is up to date in the backup process. (Refer to Retrieving File State Information in the Primary Process and Updating File State Information in the Backup Process, later in this section.)

Opening a File With a Specified Sync Depth

Normally, you use the `fopen` function to open files in the primary process. The `__ns_fopen_special` function performs the same operation as the fopen function, but it also allows a sync depth to be specified.

The sync depth is the number of nonretryable write requests that must be remembered by the opened process (server). It is used for writing operations that cannot be repeated in the backup process without changing the results of the operation. For more information about sync depth and nonretryable writes, see Updating File State Information, later in this section.

After calling `__ns_fopen_special`, the primary process calls `__ns_fget_file_open_state` to get the open state of the file, then passes the information to the backup process through interprocess communication.

Retrieving File Open State Information in the Primary Process

After opening any files required by the application, the primary process calls `__ns_fget_file_open_state` function to retrieve open state information for an open file. The primary process sends the open state information to the backup process through interprocess communication. The backup process receives the information through $RECEIVE, then calls the `__ns_backup_fopen` function to do a backup open of the file.

Opening Files in the Backup Process

The backup process opens files that have been opened by the primary process by performing a backup open. A backup open is a form of file open that permits a file to be open concurrently by both the primary and backup processes. The backup process performs a backup open of a file by calling the `__ns_backup_fopen` function.

Before doing a backup open of a file, the backup process must obtain certain file open state information from the primary process. The primary process obtains this information by calling the `__ns_fget_file_open_state` function, then passes the information to the backup process through interprocess communication.

The `__ns_backup_fopen` function does not set the file state in the backup process to the corresponding file state in the primary process. If the primary process has performed operations on a file before it has been backup opened, it is your responsibility to ensure that the file state is up to date in the backup process.

Retrieving File State Information in the Primary Process

The primary process calls the `__ns_fget_file_state` function to get the current state of a file. The primary process then sends the state information to the backup process through interprocess communication. The `__ns_fget_file_state` function does not handle key-position data from Enscribe files.

Updating File State Information in the Backup Process

The backup process reads from the $RECEIVE file the file state information sent by the primary process. The backup process then calls the `__ns_fset_file_state` function to update its memory with the file state information.

Terminating the Primary and Backup Processes

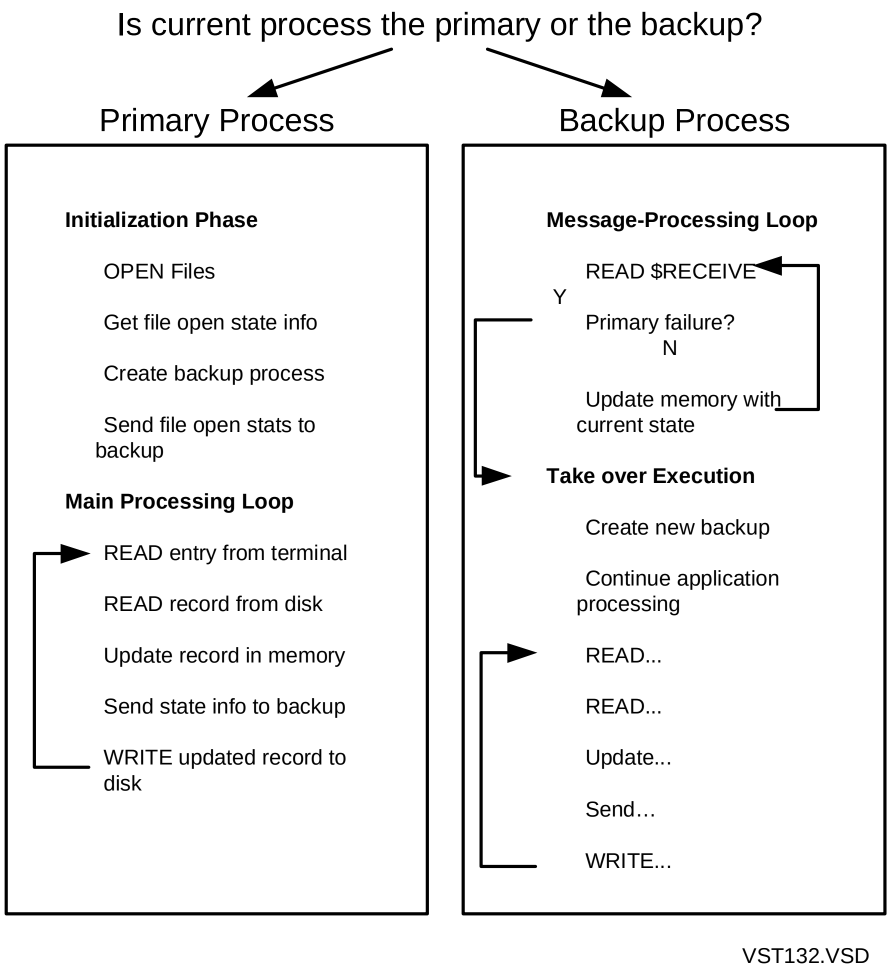
You can terminate the primary and backup processes by calling either the `exit` function or the `terminate_program` function. These functions cause both the primary and backup processes to stop when:

- The calling process is executing as a Guardian (rather than OSS) process.
- They are called from the primary process.
- Normal termination is specified. (That is, `exit(0)` or `terminate_program(0,...)` .)

Organizing an Active Backup Program

This subsection expands on the overview presented at the beginning of this section to explain how to put together an active backup program. Figure 27-1 shows a general structure for an active backup program.

Figure 27-1. Active Backup Program Structure


Is current process the primary or the backup?
Primary Process
Backup Process
Initialization Phase
OPEN Files
Get file open state info
Create backup process
Send file open stats to backup
Main Processing Loop
READ entry from terminal
READ record from disk
Update record in memory
Send state info to backup
WRITE updated record to disk
Message-Processing Loop
READ $RECEIVE
Y
Primary failure?
N
Update memory with current state
Take over Execution
Create new backup
Continue application processing
READ...
READ...
Update...
Send…
WRITE...
VST132.VSD

When an active backup program begins execution, it must first determine whether it is executing as the primary process or the backup process by calling PROCESS_GETPAIRINFO_.

The code for the primary process of a process pair consists of two phases: an initialization phase and an application processing phase. During the initialization phase, the primary process performs the tasks associated with creating and starting the backup process. During the application phase, the primary process executes the application and sends current state information to the backup process.

The code for the backup process consists of a message-processing loop, followed by an initialization phase and application processing phase. The initialization and application processing phases are essentially the same as for the primary process; that is, the backup process (once it becomes the new primary process) must also create a backup process and execute the application. The message-processing loop monitors and processes messages received from the primary process and CPU.

Primary Process Organization

A process executing as the primary process proceeds as follows:

Initialization Phase

During the initialization phase, the primary process does the following:

1. Calls `__ns_start_backup` to start and initialize the backup process, typically in another CPU. The backup process is given the same name as the primary process.
2. Opens the backup process by calling FILE_OPEN_ so that the primary process can send messages to the backup process. The messages contain the current state information.
3. Opens any files required for its execution.
4. Calls `__ns_fget_file_open_state` to get open state information for the files it just opened, and then sends the information to the backup process. The primary uses interprocess communication (for example, WRITE[X]) to write the state information to the backup process, which receives it through $RECEIVE.

The primary process can now begin processing the application.

Application Processing Phase

During application processing, the primary process does the following:

- At critical points during execution, sends updated state information to the backup process. The information includes control state, application state, and file state information. The primary process gets file state information by calling `__ns_fget_file_state`, and then uses interprocess communication (for example, WRITE[X]) to write all the state information to the backup process, which receives it through $RECEIVE.

  Note that each state update message must completely define a continuation point in the backup process.

- Monitors the backup process. If the backup process fails, the primary process should start a replacement backup. To monitor the backup process, the primary process can use one of several methods, as described later in this section under Monitoring the Backup Process.

Backup Process Organization

A process executing as the backup process proceeds as follows:

Message-Processing Loop

At the beginning of its execution, the backup process does the following:

1. Opens $RECEIVE so that it can receive messages from:
   - The operating system, indicating that the primary process or primary CPU has failed.
   - The primary process, containing current state information.
2. Calls MONITORCPUS to inform the system that the backup process is to be notified if the primary CPU fails.

   Note that if the primary process fails (rather than the CPU), the backup process is automatically notified; the backup does not need to request such notification.
3. Enters a message-processing loop in which it reads messages from $RECEIVE and takes appropriate action depending on the type of message:
   - If the message contains open state information for files opened by the primary process, the backup process calls `__ns_backup_open` to perform a backup open of the files opened by the primary process. The backup open allows files to be open concurrently by the primary and backup processes. After calling `__ns_backup_open`, the backup process continues executing the message-processing loop.

     Note that the `stderr`, `stdin`, and `stdout` files are automatically backup opened and do not require an explicit open.
   - If the message contains current state information, the backup process takes appropriate action to update its memory with the state information. For file state information, the backup process calls `__ns_fset_file_state`. For control and application state, processing is application-dependent.

     The backup process then continues executing the message-processing loop.
   - If the message indicates that the primary process or CPU has failed, the backup process takes over execution. The new primary process then exits the message-processing loop and begins the initialization phase.
   - If the message is of an unexpected type, that is, if the message is neither state information from the primary process nor an indication of process or CPU failure from the system, the backup process sends an appropriate reply. For example, such a situation occurs in the following scenario:
     1. A primary server process fails.
     2. The backup process takes over.

3. A new backup process is created within the same CPU and with the same PIN as the old primary process.
4. Before being notified of the change, a client process sends a message to what it thinks is the primary server process, but what is actually the new backup process.

The backup process checks the source of incoming messages so that it can send an appropriate reply in such cases. The backup then continues executing the message-processing loop.

Initialization and Application Processing Phases

The new primary process performs initialization and application processing activities that are essentially the same as for the original primary process:

1. Creates and starts a new backup process. The new backup process is then ready to take over if the primary process fails. The new primary process repeats the initialization steps described earlier under Primary Process Organization.

Note. The new backup process can be started in the same CPU as the original primary process or in a different CPU. The choice of which CPU to use is application-dependent.

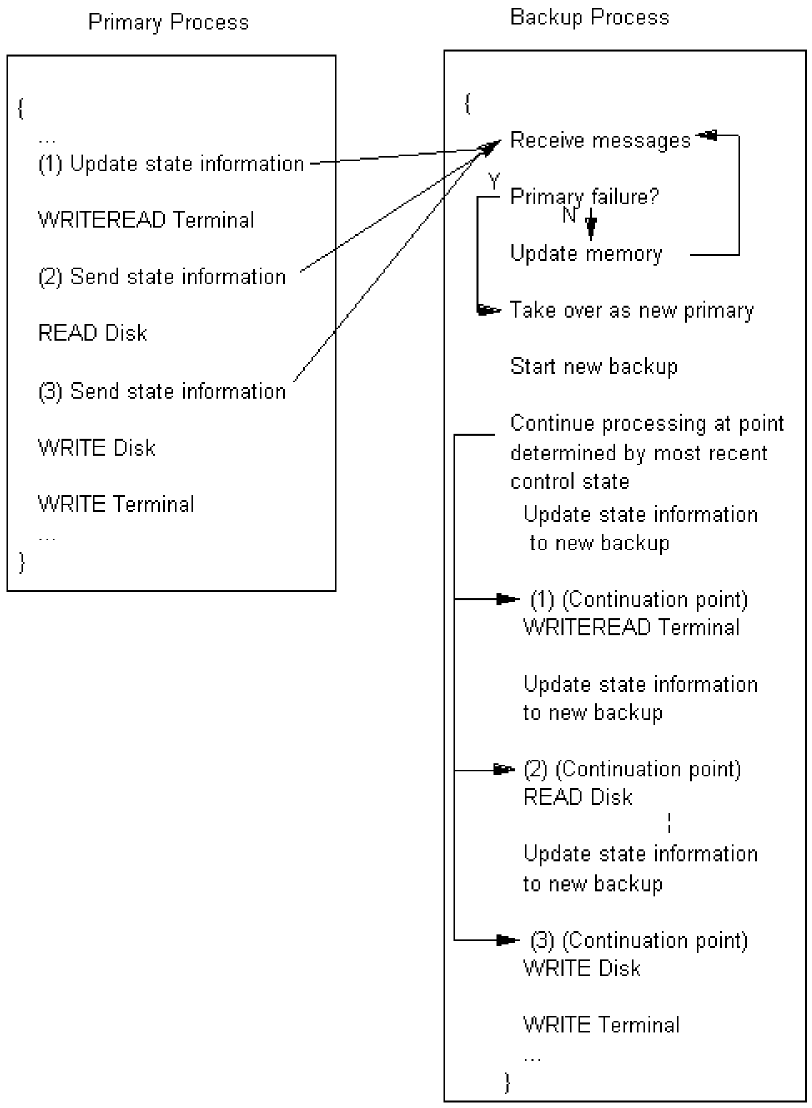
2. Resumes application processing at the point in execution indicated by the most recent state information. The new primary process sends state information to the new backup process and monitors the backup process and CPU as described above under Primary Process Organization.

Updating State Information

An important step in creating an active backup program is to develop a strategy for updating state information. State information provides the backup process with the data it needs to take over execution if the primary process fails. The information must be correct and consistent so that the backup process can continue processing without errors.

Updating state information involves saving information at a given logical point in processing and passing it to the backup process so that the backup process can take over execution at that point rather than starting at the beginning of execution. The particular information determines where execution will resume in the backup process, as illustrated in Figure 27-2. Note that for each update point in the primary process there is a corresponding continuation point in the backup process. (Note that Figure 27-2 is conceptual and does not use actual C language statements.)

Figure 27-2. Updating State Information


Primary Process
{
...
(1) Update state information
WRITEREAD Terminal
(2) Send state information
READ Disk
(3) Send state information
WRITE Disk
WRITE Terminal
...
}
Backup Process
{
Receive messages
Y
Primary failure?
N
Update memory
Take over as new primary
Start new backup
Continue processing at point determined by most recent control state
Update state information to new backup
(1) (Continuation point)
WRITEREAD Terminal
Update state information to new backup
(2) (Continuation point)
READ Disk
Update state information to new backup
(3) (Continuation point)
WRITE Disk
WRITE Terminal
...
}

VST133.VSD

The primary process sends state information to the backup process at various points during execution. Meanwhile, the backup process enters a message-processing loop in which it receives state information and failure messages. If no failure occurs, the backup updates its memory with the state information and continues in the loop. If a failure occurs, the backup takes over execution at the continuation point indicated by the most recent state information.

This subsection describes what types of information are included in the state information and gives guidelines for deciding specifically what information to update and at what points in a program's execution the information should be updated. The actual techniques and procedures you use to format, send, and receive the messages containing the state information are described earlier in this section under Organizing an Active Backup Program.

As a programmer, you must determine where to do the updates within your program and what information you want to include in each update. Enough continuation points must be provided, and each must contain enough information, so that if the primary process fails, the backup process can take over execution while maintaining the integrity of any data currently in use. Keep in mind that errors can result if you fail to include all the data that has been modified.

The number and frequency of continuation points you should provide depend on the degree of recoverability you require. As an extreme example, a primary process, after execution of each program statement, could send its entire data area to the backup process. A program of this type would be recoverable after each statement. But because of the amount of system resources needed, the program would be extremely time-consuming and inefficient.

Processes typically update only elements that have changed since the last update. This minimizes the update message length and message-handling overhead.

In developing a strategy for updating state information, you need to decide:

- What information to update
- Where within your program to place the update points
- How frequently to do the updates

Types of State Information

There are three types of state information:

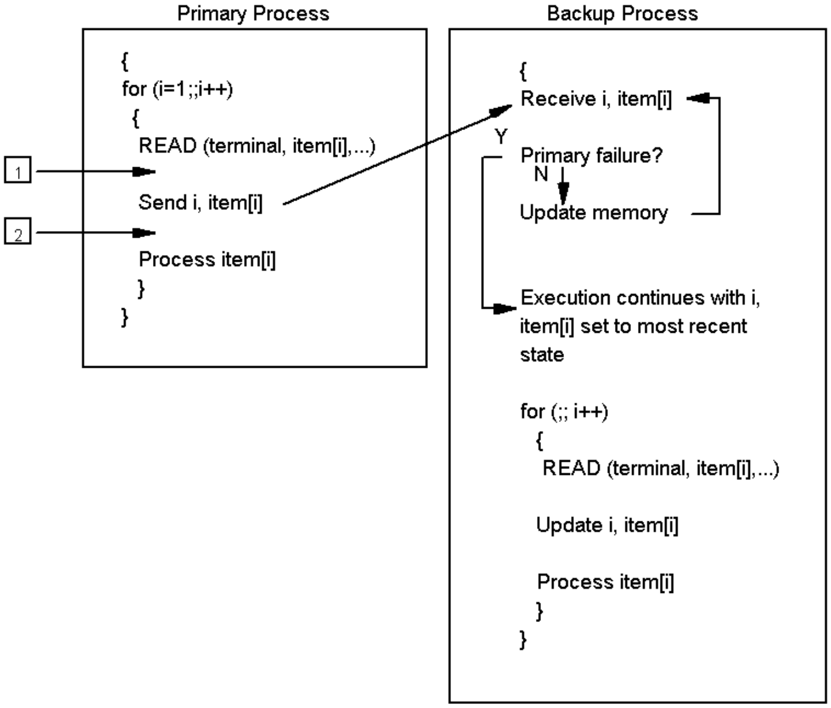
- **Control state** defines the logical points in the backup process at which execution is to resume if the primary process fails.
- **File state** consists of disk file sync blocks. Sync blocks contain control information about the current state of a disk file, including the file's sync ID. You can use the sync ID to ensure that no write operation is duplicated when the backup process takes over for the primary process.
- **Application state** gives the backup process the data values it needs to take over execution. Application state information may include file buffers and current values of process variables.

An update message from the primary process to the backup process must completely define a continuation point; that is, it must provide control state, application state, and, if I/O is done in the program, file state information.

Updating Control State Information

Control state information is used by the backup process to determine where to take over execution from the primary process. In many cases, a single scalar value is sufficient. For example, in Figure 27-3, a simple loop reads data from a terminal and processes it. (Note that Figure 27-3 is conceptual and does not use actual C language statements.)

Figure 27-3. Control State Example

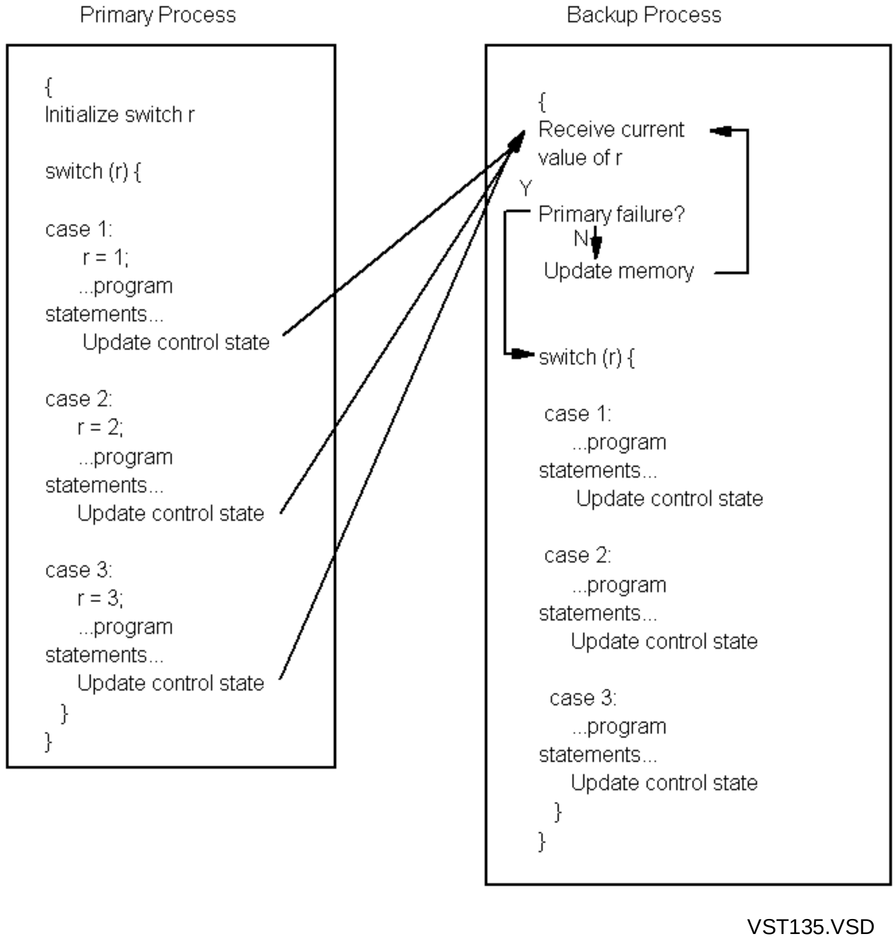

Primary Process
{
for (i=1;;i++)
{
READ (terminal, item[i],...)
1
Send i, item[i]
2
Process item[i]
}
}
Backup Process
{
Receive i, item[i]
Y
Primary failure?
N
Update memory
Execution continues with i, item[i] set to most recent state
for (;; i++)
{
READ (terminal, item[i],...)
Update i, item[i]
Process item[i]
}
}

VST134.VSD

The loop counter, i, is used to determine where to resume execution in the backup process. The update point is placed immediately after the READ. Suppose a failure occurs at point $\boxed{1}$ when i = 5. The backup process then continues execution at the previous continuation point, for which i = 4. item[4] is processed and the user must reenter item[5]. Now suppose the failure occurs at point $\boxed{2}$ while i = 5. The backup process resumes at the just-defined continuation point, processes item[5], then loops to read item[6]; in this case, the failure is transparent to the user.

Figure 27-4 shows another example of updating control state. In this example, the value of a switch is sent to the backup process. There, in the event of a takeover, the switch determines where processing is to continue by selecting the appropriate case.

Figure 27-4. Using Switch Statement to Determine the Continuation Point


Primary Process
{
Initialize switch r
switch (r) {
case 1:
r = 1;
...program statements...
Update control state
case 2:
r = 2;
...program statements...
Update control state
case 3:
r = 3;
...program statements...
Update control state
}
}
Backup Process
{
Receive current value of r
Y
Primary failure?
N
Update memory
switch (r) {
case 1:
...program statements...
Update control state
case 2:
...program statements...
Update control state
case 3:
...program statements...
Update control state
}
}
VST135.VSD

Updating File State Information

During application processing, I/O operations might be performed. At certain points in the program, the primary process must obtain file state information and send it to the backup. File state information includes file pointers and file synchronization information. File buffers are not included in the file state; they are included in the application state.

Synchronizing File Operations

File synchronization information is used by the server to determine whether an operation by a backup process after a failure of its primary process is a new operation or a retry of an operation just performed by the primary process. The information allows the server to ensure that no write operation is duplicated when a backup process takes over from its primary process. File synchronization information is contained in a file's sync block, which can be sent to the backup process at the update points. Basic file sync data can be obtained using the `_ns_fget_file_state` function. For key-sequenced files FILE_GETSYNCINFO_ is required.

The need to prevent duplicate file operations is illustrated in the following example. A primary process completes the following write operation successfully but fails before updating state information for the backup process.

```
Execution -> ...Update state information...
resumes here
                   err = POSITION (F1, -1D);  /*position to
eof*/
                   err = WRITE (F1, buff);

                   ***Primary fails here***
```

Upon taking over from the primary process, the backup process reexecutes the operations just completed by the primary process. If the WRITE were performed as requested, the record would be duplicated at the new end-of-file location.

To prevent a write operation already performed by the primary process from being duplicated by the backup process, the *`sync-depth`* parameter of the `__ns_fopen_special` function must be specified as a value greater than zero when opening the file. For a file opened in this manner, a sync ID in the sync block is used to identify the operation about to be performed by the backup process in the event of a primary process failure. If the backup process requests an operation already completed by the primary process, the server, through use of the sync ID, can recognize this condition. Then, instead of performing the requested operation again, the server returns just the completion status of the original operation to the backup process. (The completion status was saved by the system when the primary process performed the operation.) However, if the requested operation has not been performed, it is performed and the completion status is returned to the backup process. The action taken by the server is invisible to the backup process.

The server can save the completion status and reply data of the latest operations against a file and relate those completions to operations requested by a backup process upon takeover from a failed primary process. The maximum number of completion statuses that the system is to save is specified in the *`sync-depth`* parameter in the `__ns_fopen_special` function call. The *`sync-depth`* value is typically equal to the number of write operations to a file without an intervening save of the file's sync block. In most cases, the *`sync-depth`* value is 1; that is, the sync block state should be updated after each WRITE. The *`sync-depth`* value cannot exceed 15.

If the primary process fails, the backup process is notified by the operating system. The sync information received in the most recent state update message synchronizes the retry operations that the backup process is about to perform with any writes that the primary was able to complete before it failed. The backup process then retries each write in the series (in the same order as the primary process). If any operation was completed successfully by the primary process, the server does not perform the operation; instead, it just returns the completion status and data to the backup process.

The preceding example is changed to reflect the use of synchronization information:

```
Execution ----->...Update control and application state
resumes here    ...Update sync block...

                     err = POSITION (F1, -1D); /*position to
eof*/
                     err = WRITE (F1, buf);
                     ***Primary fails here***

                     Restart point
                     ...Backup takes over...
                     ...Receive sync information

                     err = POSITION (F1, -1D); /*position to
eof*/
                     err = WRITE (F1, buf);/*System detects
that*/
                                           /*operation has*/
                                           /*already been*/
                                           /*performed and*/
                                           /*returns
completion*/
                                           /*status*/
```

In this case, the write by the primary process completed successfully. On the second pass, the backup process receives a saved copy of the response made to the primary process before it failed.

Note. When buffered file operations are used, a call to a write function in your program does not necessarily correspond to exactly one write operation at the operating-system level, where the *`sync-depth`* value applies. An operating-system write operation is performed only in the following situations:

- A flush operation is performed (for example, `fflush`, `fclose`, or `exit` is called).
- The buffer is filled (buffer sizes are documented in the *C/C++ Programmer's Guide*).

See below for more information on buffered file operations.

Updating File State for Buffered File Operations

File operations can be buffered; however, the buffer is not part of the file state. This affects input and output streams differently.

For an input stream, the file position is a component of the file state. The call to `__ns_fset_file_state` updates the file position for the backup process to the file position of the primary process at the time `__ns_fget_file_state` was called. Thus, buffering is not a problem for a disk-based input stream. However, buffered but unread input from a nondisk device will be lost. Buffering can be disabled by calling the `setvbuf` or `setnbuf` function.

For an output stream, a call to a write function may leave a partially filled, unflushed buffer. The `__ns_fget_file_state` function does not cause a flush; it is your responsibility to ensure that unflushed buffers are handled appropriately. Three approaches are:

- Perform a flush operation (for example, by a call to the `fflush` function) before getting file state information.
- Ignore the effects of unflushed buffers (for situations where unflushed buffer contents are not critical to the application).
- Specify unbuffered file operations by calling the `setvbuf` or `setbuf` function.

Note that whether buffering is enabled or disabled, and whether a user-specified or system-specified buffer is used, are each specified independently and can be different in the primary and backup processes.

Updating Application State Information

Application state information is the data values needed by the backup process to take over execution from a failed primary process. This may include local variables, all or part of the data stack, and data buffers. What constitutes necessary and sufficient application state information is highly application-dependent.

Typically, file buffer state updating occurs just before writing to a disk file; the data about to be written is sent to the backup process. Careful selection of which data buffers (and corresponding file synchronization information) to send can increase the efficiency of an active backup program. An example of file buffer state updating is an entry received from a terminal: the data buffer state is updated immediately after the read to minimize the possibility that the operator would have to reenter data.

Various performance tradeoffs can be made when determining what constitutes the application state. For example, suppose an item of information can be either updated in the backup process (by the primary process sending the information to the backup process) or recomputed in the backup process on takeover. If the primary process sends the information to the backup process, the performance of the primary process is lower (because it is sending state update messages), but the time required for the backup process to take over is relatively fast. Conversely, if the backup process recomputes the state, the performance of the primary process is relatively high (because it is not sending update messages), but the performance of the backup process is relatively low (because it must recompute the state information).

Guidelines for Updating State Information

When devising a strategy for updating state information, there are two major considerations:

- The type of I/O done by the program. In general, when the backup process takes over for the primary process, repeating I/O operations ensures that they completed successfully. But certain I/O operations cannot be repeated without changing the results of the program.

- The tradeoff between recoverability and performance. The more update points a program has, the greater the degree of recoverability, but the lower the performance of the program.

Locating Update Points for Reads and Writes

The most important consideration in updating state information is to preserve the results of certain critical I/O operations. Many I/O operations cannot be repeated without changing the results of the program; therefore, you need to update file and control state to ensure that if the primary process fails, I/O operations that completed successfully are not duplicated in the backup process.

For purposes of updating state information, I/O operations can be classified as either retryable or nonretryable. A retryable operation can be repeated indefinitely with the same results. A nonretryable operation may cause erroneous or inconsistent results if repeated.

Retryable reads do not require file state updating; if the backup process takes over, it can reread the data. Most disk reads are retryable. Reads from the terminal are generally considered nonretryable. An update point should be placed immediately after each nonretryable read to protect the data just read. For reads from the terminal, this means that the user will not need to reenter the data.

Retryable writes should be repeated in the backup process to ensure that they are performed successfully. To minimize the chance of error, the continuation point should be placed immediately before the write, because at that point, the exact information to be written is known.

A nonretryable write is one that, if repeated, may cause erroneous or inconsistent results. Examples of nonretryable writes are a write to the end-of-file and the printing of forms. The sync ID can be used to detect and negate duplicate requests for nonretryable operations. Continuation points should still precede the write, but special case procedures are required to ensure consistent results. For example, a report to a line printer might need to be restarted from the last page, or a magnetic tape might need to be repositioned.

The following table summarizes the strategy for placing update points for I/O operations:

	Reads	**Writes**
Retryable	None required	Immediately before
Nonretryable	Immediately after	Immediately before, but use special case procedures

Each update point should include control state information, application state information, and, if the update precedes a write to a disk file, file state information.

Adherence to these guidelines ensures that an application program can recover from disk file operations and, in most cases, terminal operations.

Performance Versus Recoverability

In placing update points, you need to consider the tradeoff between performance and the degree of recoverability desired. For example, an application that reads and produces a summary of a file that contains hundreds of thousands of records may not require a continuation point during the read stage, because all the reads are retryable. But it might be desirable to include some degree of recovery so that, in the event of failure, it would not be necessary to repeat all the reads. On the other hand, placing an update point after every read would not be practical. A reasonable compromise might be to place an update point after every hundred, or every thousand, reads.

You should keep to a minimum the number of times you update state information in a processing loop and the amount of data in each update. But you must be sure that any update point that also defines a continuation point yields a valid program state. For example, you might update the data stack before entering a loop to ensure that the calling chain is saved, then, within the loop, update only the data that is changed within the loop.

Example of Updating State Information

The following example illustrates the placement of state information update points. The example is a simple transaction that reads data from a terminal and uses it to update a database record.

Records have the form:

```
     account_no         current_balance       credit_limit
```

They are defined by the following structure:

```
struct{
   long account_no;
   long current_balance;
   long credit_limit;
}buf2;
```

Data read from the terminal is defined as follows:

```
struct{
   long acct_no;
   long amount;
}buf1;
```

The transaction cycle is as follows:

```
err = WRITEREAD (terminal, buf1,...); /*returns acct_no
                                        and amount */
err = POSITION (account_file, buf1.acct_no);
err = READUPDATE (account_file, buf2,...);
x = buf2.current_balance + buf1.amount;
if (x > buf2.credit_limit)
   Credit limit exceeded...
else {
```

```
...Update account balance
}
```

An insufficient number of update points is added to the transaction:

```
/*First update point*/
cnt = 1;
...Update cnt (idle state)...

err = WRITEREAD (terminal,buf1,...); /*returns account_no
                                       and amount*/

/*Second update point.  Include control state and terminal
  data*/
cnt = 2;
...Update cnt, buf1...

err = POSITION (account_file, buf1.acct_no);
err = READUPDATE (account_file, buf2,...);
x = buf2.current_balance + buf1.amount;
if (x > buf2.credit_limit)
   Credit limit exceeded...
else {
   buf2.current_balance = x;
   err = WRITEUPDATE (account_file, buf2, ...);
   err = WRITE (terminal, buf1,...);
}
```

The first state update identifies the program state as being idle (or waiting for input from the terminal). The state information consists only of a counter variable set to 1. The variable is used to select the appropriate continuation point in a switch statement in the backup process.

The second state update occurs immediately after reading the terminal input. The state information consists of:

- The counter variable (to determine where to resume execution in the backup process)
- The data read from the terminal

The assumption is that, because the transaction is driven by the data read from the terminal, that data is sufficient for the backup process to perform the identical operation. This assumption is incorrect, however. A problem arises if a failure occurs just after the WRITEUPDATE of the `account_file`. The problem is illustrated in the following transaction:

```
err = WRITEREAD (terminal,buf1,...); reads account_no = 12345,
                                          amount = $10

/*Second state information update. Include control state and
terminal data*/
cnt = 2;
...Update cnt, 12345, $10...
```

```
err = POSITION (account_file, 12345D);
err = READUPDATE (account_file, buf2,...);   returns the following:
```

account_no	current_balance	credit_limit
12345	$485	$500

```
x = $485 + $10;
if (x > $500) ...
current_balance = x;
err = WRITEUPDATE (account_file, buf2, ...);  writes the following:
```

account_no	current_balance	credit_limit
12345	$495	$500

```
*********** FAILURE OCCURS HERE**********
```

The backup process resumes with the latest application state information: `account_no = 12345` and `amount = $10`.

```
case (cnt = 2):
err = POSITION (account_file, 12345D);
err = READ (account_file, buf2,...);  reads the following:
```

account_no	current_balance	credit_limit
12345	$495	$500

```
x = $495 + 10;
IF (x > $500)...
```

Here, the test fails because the update to disk completed successfully and `current_balance` has already been updated. The user is given an indication that account number 12345 has exceeded its credit limit; therefore, the purchase is refused. However, the balance in account 12345 reflects that a purchase was made.

An additional update point is now added to the transaction cycle:

```
/*First update point*/
cnt = 1;
...Update cnt (idle state)...

err = WRITEREAD (terminal,buf1,...); /*returns account_no
                                       and amount */

/*Second update point. Include control state and*/
/*terminal data */
cnt = 2;
...Update cnt, buf1...

err = POSITION (account_file, account_no);
err = READUPDATE (account_file, buf2,...);
x = buf2.current_balance + buf1.amount;
if (x > credit_limit)
   credit limit exceeded...
else
   buf2.current_balance = x;
```

```
/*Third update point. Include control state (cnt), data
  state (buf2), and file state (account_file)*/
cnt = 3;
...Update cnt, buf2, account_file...

err = WRITEUPDATE (account_file, buf2, ...);
err = WRITE (terminal, buf1,...);
```

The third update point identifies the program state as “preparing to write an updated record to disk.” The state information consists of:

- The counter variable (control state)
- The updated record (data state)
- The disk file’s sync information (file state)

If the primary process fails between update points 1 and 2, the backup process reissues the WRITEREAD to the terminal. If the primary process fails between update points 2 and 3, the backup process uses the terminal input and continues processing the transaction. If the primary process fails after update point 3, the backup process uses the current state information to reexecute the write to disk.

Note that update point 2 could be omitted. If this were done, a failure between update points 2 and 3 would require the operator to reenter the transaction.

Saving State Information for Multiple Disk Updates

When performing a series of updates to one or more disk files, you can save state information for those updates at one point in the program instead of multiple points. This results in lower system usage.

The program should be structured so that the series of writes needed to update a file are performed in a group. For each file to be updated in this manner, you should specify the *sync-depth* parameter of the FILE_OPEN_ procedure as the maximum number of calls to the WRITE[X] procedure that are made between points at which state information is updated. Then, just before performing *sync-depth* writes to the file, update the state information, including the file’s sync block and the data buffers about to be written to the file.

Providing Communication Between the Primary and Backup Processes

Active backup programs require a method for communication between the primary and backup processes and between the backup process and the operating system. HP recommends using the Guardian interprocess communication facility for:

- Sending state information from the primary process to the backup process.
- Receiving state information in the backup process.

- Receiving system status messages in the backup process. The system messages tell the backup process when the primary process or CPU has failed.

This subsection gives a brief overview of communication between the primary and backup processes. The procedures of the Guardian interprocess communication facility are described in detail in the *Guardian Procedure Calls Reference Manual*. The use of those procedures for communicating between processes is described in Section 6 of this manual, Communicating With Processes.

Your program can use the FILE_OPEN_, WRITE[X], and READ[X] procedures in the following way:

1. The primary process calls FILE_OPEN_ to open the backup process.
2. The primary process calls WRITE[X] to send state information messages to the backup process.
3. The backup process calls FILE_OPEN_ to open the $RECEIVE file.
4. The backup process calls READ[X] to read status and state information messages from the $RECEIVE file. The $RECEIVE file contains the state information messages sent by the primary process and execution status messages sent by the system.

You can implement a protocol in which the backup process replies to the primary process by using the WRITEREAD[X], READUPDATE[X], and REPLY[X] procedures.

As explained later in this section, you can also use nowait I/O to allow the primary process to check that the backup process has not failed.

Sending Messages From the Primary to the Backup

To send a message to the backup process, the primary process follows these steps:

1. Opens the backup process
2. Creates the message in a buffer
3. Sends the message to the backup process and, optionally, waits for a reply

To open the backup process, the primary process calls the FILE_OPEN_ procedure. For example:

```
{
...Get backup process name and name length...
error = FILE_OPEN_(process_name, /*backup process name*/
                  process_name_len, /*name length*/
                  &backup_filenum); /*backup process file*/
                                    /*number returned*/
}
```

The FILE_OPEN_ procedure returns the file number of the backup process.

The time at which the open finishes depends on the way the backup process opens $RECEIVE. See Section 6, Communicating With Processes, for details.

Once the backup process is open, the primary process can communicate with the backup process by writing messages to the file number returned by the FILE_OPEN_ call. To send a message, the primary process can use either WRITE[X] (for one-way communication only) or WRITEREAD[X] (for one-way or two-way communication).

The following example sends a state message to the backup process without expecting a reply:

```
{
...
error = WRITEX (backup_filenum, /*file number of backup*/
               msg_buffer, /*Buffer containing state info*/
               length, /*length of state info*/
               count_written); /*number of bytes read by
                                 backup*/
...
}
```

Receiving Messages in the Backup Process

To receive messages from the primary process or the system, the backup process first opens the $RECEIVE file, then reads messages from it. For two-way communication, in which the backup process replies to the primary process, set the *`receive-depth`* parameter of the FILE_OPEN_ procedure to a value greater than zero. For one-way communication, in which the backup process does not reply to the primary process, set the *`receive-depth`* parameter to zero.

The first word of a system message is always a negative message number. The primary process can insert a positive number in the first word of a state information message, thereby giving the backup process a means of distinguishing between system messages and state information messages.

The following example opens and reads from $RECEIVE for one-way communication. The program tests the first word of the message to determine whether it is a system message or a state information message:

```
/*Open $RECEIVE*/
filename = "$RECEIVE";
length = 8;
error = FILE_OPEN_(filename, /*the $RECEIVE file*/
                   length, /*length of filename*/
                   &filenum, /*file number returned*/
                   /*access*/
                   /*exclusion*/
                   /*nowait*/
                   /*receive_depth*/);

/*Read and process message*/
err = READX (filenum, /*file number of $RECEIVE file*/
             msg_buffer, /*message returned*/
             count_read, /*bytes read*/);

/*Process message based on message number*/
```

```
switch (msg_buffer.number)
{
   case ... /*Processor failure*/
   case ... /*Process deletion*/
      ...Backup takes over for primary...
   case ... /*State information*/
      ...Update control state ...
     ... Update application state ...
     ... Update file state ...
}
```

Monitoring the Backup Process

When writing an active backup application, it is advisable to structure the primary process to periodically check the backup process to ensure that it is running and ready to take over in the event of a primary process failure. Some ways of monitoring the backup process are discussed below.

Calling PROCESS_GETPAIRINFO_

You can monitor the backup process by calling the PROCESS_GETPAIRINFO_ procedure. This procedure returns the process handle of the backup process; if the backup process does not exist (has terminated due to process or CPU failure, the process handle has a null value). The examples at the end of this section show a user-written C function that calls PROCESS_GETPAIRINFO_ to monitor the backup process.

Using Nowait I/O

An alternative method for detecting a failed backup process or CPU is to use NOWAIT I/O as follows:

- Open the backup process for nowait I/O (by specifying the appropriate parameter in the FILE_OPEN_ call).
- When performing write operations to the backup process (for updating state information), call AWAITIO[X] with a positive value for the `timelimit` parameter. If the write to the backup process does not finish within the specified time limit, the primary process can perform further checking to determine whether the backup has actually failed.

Checking I/O Error Status

You can monitor the backup process simply by checking the error status code after an I/O operation that sends state information to the backup process. If the backup process no longer exists, an error condition is returned.

Reading $RECEIVE

You can monitor the backup process by reading the $RECEIVE file in the same manner as the backup process. The operating system sends a message to $RECEIVE if the backup process fails.

Programming Considerations

Following are some general considerations for writing active backup programs in C.

Compile-Time and Linker Considerations

For TNS and accelerated programs, compiler and Binder issues are:

- The `nonstoph` header file contains the declarations that support active backup programming. (The functions that support fault-tolerant programming are implemented in TAL. By using the `nonstoph` header, the appropriate TAL interface code is generated.) Functions declared in the `nonstoph` header are defined in the `cnonstop` library file.
- HP provides two different file-reference models: the ANSI model and the alternate model. Only the ANSI model supports fault-tolerant file operations (for example, open with sync depth, backup open, get file state, and so on).
- EDIT files do not support fault-tolerant file operations. Therefore, the `ANSISTREAMS` pragma must be specified during compilation of the main function so that the standard files will be opened as ANSI files (code 180 files) instead of EDIT files.
- Active backup programs must use the large-memory model. The large-memory model uses 32-bit addressing and supports the wide-data model. Refer to the *C/C++ Programmer’s Guide* for additional information about the large-memory model.
- For application portability and compatibility with future software releases, HP recommends that active backup programs use the wide-data model. In the wide-data model, the data type `int` is 32-bits.
- Active backup C programs can include TAL code, but the TAL components of an application cannot use passive backup programming techniques. Other mixed-language programming is not allowed.
- An active backup program can be accelerated.
- An active backup program must run in the Common Run-Time Environment (CRE).
- An active backup program can run only as a Guardian process; it cannot run as an OSS process.

For TNS/R native programs, compiler and linker issues are:

- The `crtlnsh` header file contains the declarations that support active backup programming. (The functions that support fault-tolerant programming are implemented in pTAL. By using the `crtlnsh` header, the appropriate TAL interface code is generated.) Functions declared in the `crtlnsh` header are defined in the `crtlns` library file.
- HP provides two different file-reference models: the ANSI model and the alternate model. Only the ANSI model supports fault-tolerant file operations (for example, open with sync depth, backup open, get file state, and so on).
- EDIT files do not support fault-tolerant file operations. Therefore, the `ANSISTREAMS` pragma must be specified for compilation of the main function so that the standard files will be opened as ANSI files (code 180 files) instead of EDIT files.
- Active backup C programs can include pTAL code, but the pTAL components of an application cannot use passive backup programming techniques. Other mixed-language programming is not allowed.
- An active backup program must run in the Common Run-Time Environment (CRE).
- An active backup program can run only as a Guardian process; it cannot run as an OSS process.

Run-Time Considerations

Run-time issues are:

- An active backup program can use the full range of memory management facilities; no special support is needed. The primary and backup processes never need to have the same memory state, and each process manages its memory independently.
- The standard C files (`stdin`, `stdout`, and `stderr`) are automatically backup opened (in the backup process). Any other files on which fault-tolerant operations will be performed must be explicitly backup opened by the `__ns_backup_open` function.
- Errors detected in the backup process are not automatically communicated to the primary process; user code must be written to handle such processing.

Comparison of Active Backup and Passive Backup

For the benefit of programmers familiar with the passive backup programming techniques supported by TAL, this subsection provides an overview of the differences between active backup programming and passive backup programming.

Active backup differs from passive backup in that active backup does not have Guardian procedures equivalent to the various passive backup CHECK* routines. The active backup programmer must explicitly provide for this function through various coding techniques and use of C functions described later in this section.

In active backup programming, the programmer must do more explicit programming than in passive backup programming. However, active backup programming provides the following advantages:

- Active backup programs can have better performance, because their fault-tolerant functions can be specifically tailored to the application.
- It is potentially easier to convert ported reusable code components and applications to run in a fault-tolerant manner, because any hidden state information need not be sent to the backup process.

Table 27-1 summarizes the differences between active backup programming and passive backup programming. Native processes cannot call CHECKPOINT[MANY]; they must call CHECKPOINT[MANY]X instead.

Table 27-1. Differences Between C Active Backup and TAL Passive Backup (page 1 of 2)

To Perform This Task	An Active Backup Program Does This	A Passive Backup Program Does This
Start backup process	Primary calls __ns_start_backup.	Primary calls PROCESS_CREATE_.
Establish communication between primary and backup processes	Primary opens backup, and backup opens $RECEIVE. Protocol is application-specific, implemented with Guardian read and write routines.	Backup calls CHECKMONITOR; primary calls CHECK* routines. Protocol defined by the various CHECK* routines.
Monitor primary CPU by backup process	Backup calls MONITORCPUS; backup is explicitly coded to check for failure messages.	Backup calls MONITORCPUS. CHECKMONITOR checks for failure messages.
Monitor backup CPU by primary process	Primary can call PROCESS_GETPAIRINFO_ or read messages from $RECEIVE or check for failures when communicating with backup.	Same as active backup (explicit code needed).
Backup open files for the backup process	Primary calls __ns_fget_file_open_state, sends open state to backup. Backup calls __ns_backup_fopen.	Primary calls FILE_OPEN_CHKPT_.
Retrieve file state information and send it to backup	Primary calls __ns_fget_file_state and writes state information to backup.	Primary calls CHECKPOINT[MANY][X].
Set file state in the backup process	Backup calls __ns_fset_file_state after receiving file state info from primary.	Done automatically while backup is running in CHECKMONITOR.
Define continuation points	Primary sends control state information to backup.	Continuation point defined by location of most recent CHECKPOINT[MANY][X] call.

Table 27-1. Differences Between C Active Backup and TAL Passive Backup (page 2 of 2)

To Perform This Task	An Active Backup Program Does This	A Passive Backup Program Does This
Send data state information to backup process	Primary sends data state to backup through interprocess communication. Backup must update its own memory.	Primary calls CHECKPOINT[MANY][X].
Define content of data state information	Application-dependent.	All of memory.
Implement message-processing loop in backup, process messages, and take over execution if primary fails	Explicitly coded by programmer.	Initiated by the call to CHECKMONITOR.

Active Backup Example 1

A simple example is now used to illustrate how to structure an active backup program. The following application consists of a loop that increments a counter.

```
/*This function does the work of the application, which is
  to increment a counter.  To avoid overflow, the
  application wraps around after 10,000 iterations*/

int main (void)
{
   short counter;

   for (;;)
   {
      counter++;
      if (counter > 10000) counter = 0;
      /*Code that uses the counter value would go here*/
   }
}
```

The rest of this subsection shows the steps involved in creating an active backup version of this program. The program is coded in a modular style, with the various active backup activities written as separate functions. You can use this example as a template for writing your own active backup applications.

To simplify the presentation, error conditions returned by C library routines and Guardian procedures are not handled.

Figure 27-5 and Figure 27-6 show the functional flow for the primary and backup processes, respectively. The names outside the boxes indicate user-written functions that perform the activities.

Figure 27-5. Primary Process Functional Flow

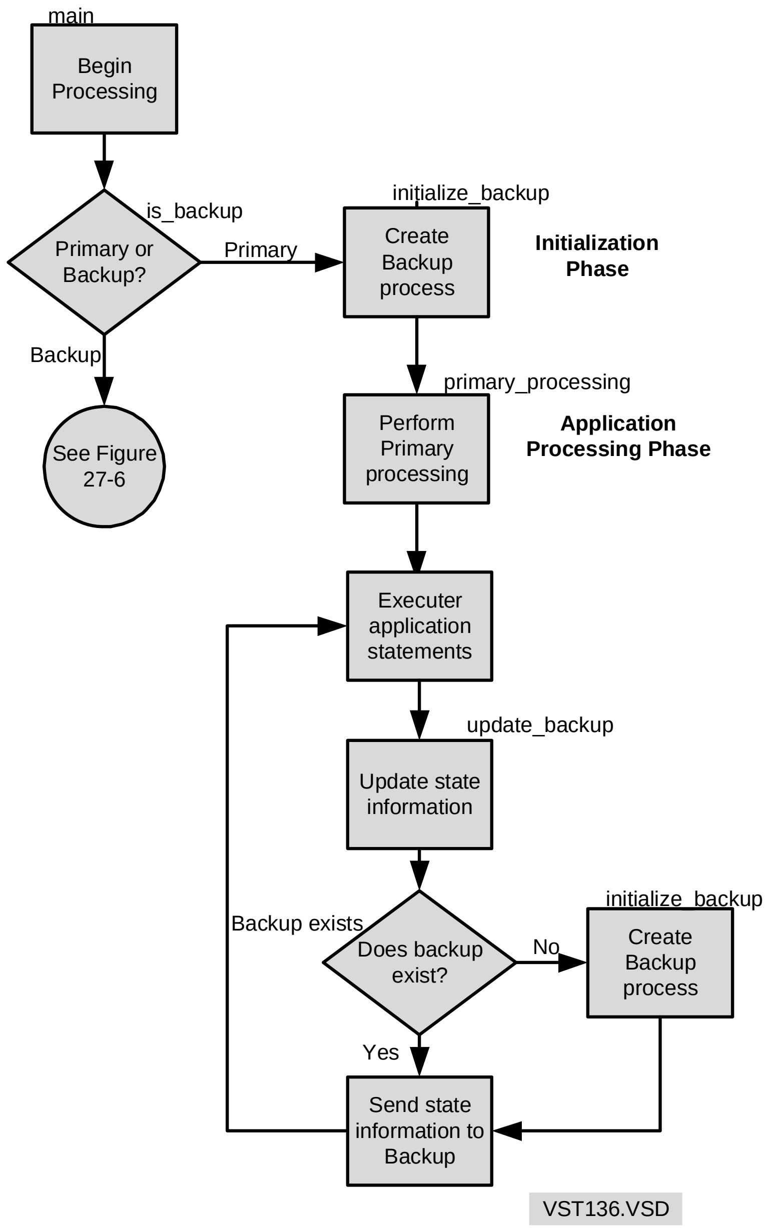
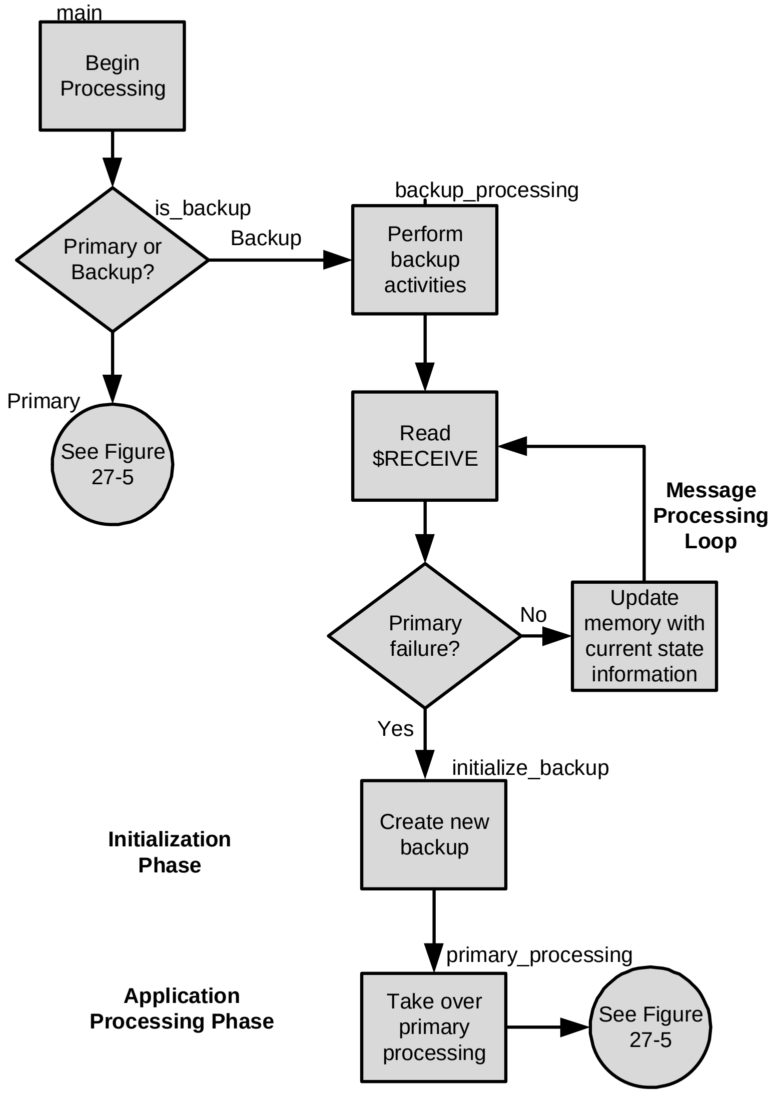

main
Begin Processing
is_backup
Primary or Backup?
Primary
Backup
See Figure 27-6
initialize_backup
Create Backup process
Initialization Phase
primary_processing
Perform Primary processing
Application Processing Phase
Executer application statements
update_backup
Update state information
Backup exists
Does backup exist?
No
initialize_backup
Create Backup process
Yes
Send state information to Backup
VST136.VSD

Figure 27-6. Backup Process Functional Flow


main
Begin Processing
is_backup
Primary or Backup?
Backup
backup_processing
Perform backup activities
Primary
See Figure 27-5
Read $RECEIVE
Message Processing Loop
Primary failure?
No
Update memory with current state information
Yes
initialize_backup
Initialization Phase
Create new backup
primary_processing
Application Processing Phase
Take over primary processing
See Figure 27-5

VST137.VSD

Program Declarations

The particular declarations required by an active backup program depend on the application. The following are used in this example.

#include and #pragma Lines

The following `#include` and `#pragma` lines provide the declarations required by the program. The conditional statement `#ifdef _TNS_R_TARGET` causes the lines that follow it to be included only for native programs, while the lines that are between the `#else` line and the `#endif` line are included only for TNS and accelerated programs. For native programs, the pragmas `search "$system.system.crtlns"` and `runnamed` must be included in the compiler command line.

```
#ifdef _TNS_R_TARGET
/* for native programs only */
#include <crtlnsh> nolist

/*  Note: Include the pragmas 'search "$system.system.crtlns"'
*/
/*  and "runnamed" in the compiler command line for
*/
/*  native programs.
*/

#else
/* for TNS and accelerated programs only /*
#pragma wide
#pragma search "$system.system.cnonstop"
#include <nonstoph> nolist
#endif

#include <stringh> nolist
#include <stdioh> nolist

#include <talh> nolist

#include <cextdecs (FILE_GETRECEIVEINFO_,
                    FILE_OPEN_,
                    MONITORCPUS,
                    PROCESS_GETINFO_,
                    PROCESS_GETPAIRINFO_,
                    PROCESS_SETINFO_,
                    PROCESSHANDLE_COMPARE_,
                    PROCESSHANDLE_DECOMPOSE_,
                    READUPDATEX,
                    REPLYX,
                   WRITEX)> nolist

#include "$system.zsysdefs.zsysc
(zsys_ddl_phandle,zsys_ddl_receiveinformation)" nolist

#include "$system.zspidef.zfilc (constants)" nolist
```

`#include "$system.zspidef.zspic (zspi_ddl_char8,zspi_ddl_unit)" nolist`

`#pragma search "$system.system.cnonstop"` for TNS and accelerated programs, or `search "$system.system.crtlns"` in the compiler command line for TNS/R native programs, causes the library containing the C functions for fault-tolerant support to be bound into the program.

`#pragma wide` specifies the wide-data model (TNS and accelerated programs only).

`nonstoph` for TNS and accelerated programs, or `crtlnsh` for native programs, provides declarations required by the C functions for active backup programming. `stringh` and `stdioh` are standard header files. `cextdecs` includes declarations for the following Guardian procedures required by the program:

- FILE_GETRECEIVEINFO_, which is called by the backup process to get information about messages it receives.
- FILE_OPEN_, which is called by the primary process to open the backup process, and by the backup process to open $RECEIVE.
- MONITORCPUS, which is called by the backup process to monitor the primary process CPU. The operating system sends the backup process a “processor down” message if the monitored CPU fails.
- PROCESS_GETINFO_, PROCESS_GETPAIRINFO_, and PROCESSHANDLE_DECOMPOSE_, which are used to obtain information about the primary and backup processes.
- PROCESSHANDLE_COMPARE_, which is called by the backup process to determine if a message it received was sent by the primary process.
- READUPDATEX, which is called by the backup process to receive messages from the primary process through $RECEIVE.
- REPLYX, which is called by the backup process to reply to messages it receives.
- WRITEX, which is called by the primary process to send messages to the backup process.

Message Declarations

The program uses four message formats: two defined by the operating system and two defined by the program:

- CPU_DOWN is a system message that indicates that a monitored CPU is down. This message is used to notify the backup process, if the primary process CPU fails, so that the backup process can take over processing.
- PROCESS_DELETION is a system message that indicates that a process has terminated. This message is used to notify the backup process, if the primary process terminates, so that the backup process can take over processing.

- UPDATE_MESSAGE is an application-defined message that contains the current state information. In this example, the value of the counter serves as both application state and control state information: it is the data needed by the backup process to continue processing (application state), and it defines the point at which processing is to resume (control state). The primary process passes this value to the backup process, which updates its state information.
- IGNORE_MESSAGE is an application-defined message number that, for internal processing, is substituted for the number of a message that has been received from a source other than the primary process or the system. After a reply is sent with an indication of OWNERSHIP_ERROR, no further action is taken on such a message.

The message declaration portion of the program serves the following purposes:

- Provides macros that define the message numbers. Note that the system message numbers are negative and the application-defined message numbers are positive.
- Declares a structure for each of the three message formats. The first member of the structure contains the message number, and the second member is a union with one member per message format. The information contained in the system messages is not used (only the message number is used), so the exact structure is not defined.

Two other elements are defined that are used by the backup process if it receives a message it does not expect: IGNORE_MESSAGE is used as a message number and is placed in the message number portion of the message structure, and OWNERSHIP_ERROR is an error value that is returned to the sender.

The message declarations are:

```
/*message numbers*/
#define CPU_DOWN -2              /*system message*/

#define PROCESS_DELETION -101    /*system message*/
#define UPDATE_MESSAGE 1         /*application message*/

#define IGNORE_MESSAGE 0         /*application message*/
#define OWNERSHIP_ERROR 200      /*used by backup*/

#define FEOWNERSHIP 200

#define IS_BACKUP 6      /* Returned from PROCESS_GETPAIRINFO_*/


#define RECEIVE_FILENAME "$RECEIVE"

/* Maximum file name length (plus 1 to allow for null
terminator)*/

#define MAXNAMELEN 36

/* Process handle size and null process handle. A phandle is 10
words long; a null phandle has all 10 words set to -1 */
```

```
#define PHANDLESIZE 10

#define NULLPHANDLE {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

/* Primary process attribute code for PROCESS_SETINFO_ */

# define PRIMARY_ATTRIBUTE 47

/* File number of the backup process. Used by the primary
process to send information to the backup */

short backup_filenum;


/* Global data */

short counter = 0;

/* Variable for use by the backup to hold the counter state
value sent by the primary */

short counter_state;

/*The following structure defines the various message formats
used by this program.*/

typedef struct
{
   short msgnumber; /*message number*/
   union

   {

      /*one member for each message format*/
      /*UPDATE_MESSAGE message*/
      short counter_value;

      /*CPU_DOWN message*/
      short cpu_down_info;

      /*PROCESS_DELETION message, variable-length fields
      are assumed to have zero length*/
      short process_death_notification_info [40];

   } msg_variants;
} message_format;
```

Miscellaneous Declarations

Finally, the program contains miscellaneous declarations to define the following:

- A value to be returned by the PROCESS_GETPAIRINFO_ procedure, used to determine whether the process is running as the primary or backup process.
- The $RECEIVE file name.
- The maximum file-name length.
- The size of the process handle, and a null process handle. A process handle is 10 words long; a null process handle has all ten words set to -1.
- The file number of the backup process. This number is used by the primary process to send information to the backup.
- A variable to be used by the backup process to hold the counter state value sent by the primary process.

The declarations are as follows:

```
#define RECEIVE_FILENAME "$RECEIVE"

/*Maximum file name length (plus 1 to allow for null
   terminator)*/
#define MAXNAMELEN 36

/*Process handle size and null process handle. A phandle is 10
words long; a null phandle has all 10 words set to -1*/
#define PHANDLESIZE 10
#define NULLPHANDLE {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}

/*Primary process attribute code for PROCESS_SETINFO_*/
#define PRIMARY_ATTRIBUTE 47

/*File number of the backup process. Used by the primary process
to send information to the backup*/
short backup_filenum;

/*Global data*/
short counter = 0;

/*Variable for use by the backup to hold the counter state
   value sent by the primary*/
short counter_state;

short primary_phandle [PHANDLESIZE];

short backup_phandle [PHANDLESIZE];
```

Creating and Starting the Backup Process

The primary process performs several activities related to initializing the backup process. Specifically, it starts the backup process and opens it for interprocess communication so that the primary process can send messages to the backup process. The operating system is allowed to select the CPU in which the backup process will run. It is convenient to code these activities as separate functions, as follows:

- Function `initialize_backup`, which opens and starts the backup process; called by function `primary_processing`.
- Function `primary_cpu`, which returns the CPU number of the primary process CPU. The CPU number is required by the MONITORCPUS procedure.

Function initialize_backup

Function `initialize_backup` performs the initialization tasks of the primary process: it starts and opens a backup process so that the primary process can send messages to the backup process.

To start the backup process, `initialize_backup` calls the C function `__ns_start_backup` with the input parameter -1, which specifies that the operating system is to select the backup CPU. `__ns_start_backup` returns the backup process handle. `initialize_backup` then calls FILE_OPEN_ to open the backup process.

```
/* This function is called by the primary process to
   initialize the backup process.  It starts the backup
   process and opens it for interprocess communication */
void initialize_backup (void)
{
   char process_name [MAXNAMELEN];
   short process_name_len;
   short error;
   short error_detail;

   /*Start the backup process*/
   error = __ns_start_backup (&error_detail, (short) -1,
                              backup_phandle);

   /*Get the process name of the backup process*/
   error = PROCESSHANDLE_DECOMPOSE_ (backup_phandle,
                                    /*cpu*/,
                                    /*pin*/,
                                    /*nodenumber*/,
                                    /*nodename:nmax*/,,
                                    /*nlen*/,
                                    process_name,
                                    MAXNAMELEN,
                                    &process_name_len);

   /*Open backup process for interprocess communication*/
   error = FILE_OPEN_ (process_name, process_name_len,
                       &backup_filenum);
}
```

Function primary_cpu

This function returns the CPU number of the primary process CPU.

```
/*This function returns the processor number of the primary
   process processor*/
short primary_cpu (void)
{
   short cpu;
   short error;

   /*Get the primary process phandle*/
   error = PROCESS_GETPAIRINFO_ (/*current process*/,
                                 /*pair:maxlen*/,,
                                 /*pairlen*/,
                                 primary_phandle);

   /*Get and return the primary process processor*/
   error = PROCESSHANDLE_DECOMPOSE_ (primary_phandle,
                                      &cpu);
return cpu;
}
```

Updating State Information

After the primary process performs its initialization tasks, it begins executing the main processing loop. Code is added to the main processing loop to update state information in the backup process. The state information includes control state and application state; file state is not needed because the program does no I/O. The value of the loop counter provides control state and application state: it tells the backup process where to resume execution, and it is the data needed to continue application processing. File state information for the `stdout` file is updated.

The code to update state information is written as a separate function, named `update_backup`, which is called by the primary process.

Function backup_exists

Function `backup_exists` tests for the existence of the backup process by calling PROCESS_GETPAIRINFO_. If the backup does exist, the function returns -1. This function is called by `update_backup` to make sure that the backup process exists before updating state information.

```
short backup_exists (void)
{
   short null_phandle [PHANDLESIZE] = NULLPHANDLE;
   short error;
   error = PROCESS_GETPAIRINFO_ (/*current process*/,
                                 /*pair:maxlen*/,,
                                 /*pairlen*/,
                                 /*primary_phandle*/,
                                 backup_phandle);
   return (short) memcmp (backup_phandle, null_phandle,
           PHANDLESIZE * 2);
}
```

Function update_backup

Function `update_backup` is called by the primary process to update state information in the backup process. To update the state information, the primary process must send a message to the backup process that contains the value of the loop counter.

Function `update_backup` does the following:

- Checks for the existence of the backup process. If the backup process does not exist, an attempt is made to recreate it by calling `initialize_backup`. The code to check for the existence of the backup process is written as a separate function, named `backup_exists`.
- Creates the update message and sends it to the backup process through interprocess communication (WRITEX).

Function `update_backup` is as follows:

```
/*This function is called by the primary process to update
  the state of the backup process.  It creates and sends a
  message to the backup process.  Parameter i is the counter
  value.*/
void update_backup (short i)
{
   _cc_status_short cond_code;    /*value not used in this
example*/
   message_format message;

   /*Check for existence of backup, attempt to recreate
      if necessary*/
   if (!backup_exists())
      initialize_backup();

   /*Create the update message*/
   message.msgnumber = UPDATE_MESSAGE;
   message.msg_variants.counter_value = i;

   /*Send the update message to the backup*/
   cond_code = WRITEX (backup_filenum,
                       (char *) &message,
                       (short) sizeof message);
   /*When checking the condition code, include a check for
     the existence of the backup process*/
}
```

Primary and Backup Processing

The original application program can now be restructured into the following three functions:

- Function `primary_processing`, which performs the primary process activities: it does the work of the application and updates state information in the backup process.
- Function `backup_processing`, which performs the backup process activities: it receives state information from the primary process and monitors the primary process CPU. If a message indicating CPU failure or process termination is received, the backup process takes over execution.
- Function `main`, which is where execution of the active backup program begins. Function `main` determines whether it is running as the primary process or the backup process: if it is the primary process, it initializes the backup process and calls `primary_processing`; if it is the backup process, it calls `backup_processing`.

  A separate function, `is_backup_process`, is written to determine whether the program is running as the primary process or the backup process. `main` calls this function at the beginning of its execution.

Function primary_processing

Function `primary_processing` does the work of the application (which, in this example, is to execute an infinite loop that increments a counter). Within the innermost loop, function `update_backup` is called to update the state information in the backup process.

```
/*This function does the work of the application, which is
  to increment a counter.  To avoid overflow, the
  application wraps around after 10,000 iterations*/

void primary_processing (void)
{
   for (;;)
   {
      counter++;

      printf ("Counter is %d\n", counter);
      if (counter > 10000) counter = 0;
      update_backup (counter);
   }
}
```

Function backup_processing

Function `backup_processing` handles the backup processing tasks, which include:

- Open $RECEIVE so that the backup process can receive messages from the primary process and the operating system.
- Call MONITORCPUS to indicate to the operating system that the backup process is to be notified if the primary process CPU fails.
- Enter an infinite loop that reads and processes messages from $RECEIVE.

The three message types are processed as follows:

- CPU_DOWN and PROCESS_DELETION indicate that the primary process has failed. The backup process takes over execution and becomes the primary process, then does the following:
  - Initializes a new backup process (by means of `initialize_backup`).
  - Sets the counter value to the most recent state information.
  - Calls `primary_processing` to continue as the primary process.
- UPDATE_MESSAGE contains the counter value (the control state and application state information). The backup process saves the counter value in the `counter_state` variable.

Statements are included to test the process handle of the sender of the incoming message and to return an appropriate response if the sender is not the primary process or the system. If the message is from an unexpected sender, the backup process replies with ZFIL_ERR_OWNERSHIP. This triggers a process-handle refresh in the sender, which can then retry sending the message.

Function `backup_processing` is as follows:

```
/*This function performs processing for the backup process.
  It opens $RECEIVE, monitors the primary processor, and
executes a loop that receives messages through $RECEIVE.*/

void backup_processing (void)
{
   short receive_filenum;
   message_format message;
   short countread;
   zsys_ddl_receiveinformation_def receiveinfo;
   short primary_value = 1;
   short counter;
   _cc_status cond_code;
   short error;

   short primary_phandle [PHANDLESIZE];

   short null_phandle [PHANDLESIZE] = NULLPHANDLE;
```

```
/* open $RECEIVE */

error = FILE_OPEN_ (RECEIVE_FILENAME,
                    (short) strlen (RECEIVE_FILENAME),
                    &receive_filenum,
                    /*access*/,
                    /*exclusion*/,
                    /*nowait*/,
                    1 /*receive depth*/);

/*Monitor the primary process processor.  The backup receives
  a message if the primary process goes down*/

MONITORCPUS ((unsigned short)(0x8000 >> primary_cpu ()));
/*Infinite loop that receives and processes messages*/
for (;;)
{
   /*Read the next message from $RECEIVE with the intention
     of replying*/

   cond_code = READUPDATEX (receive_filenum,
                            (char *) &message,
                            (short) sizeof (message_format),
                            &countread);

   /*Check process handle of sender of incoming message*/

   FILE_GETRECEIVEINFO_ ((short *) &receive_info);

   error = PROCESS_GETPAIRINFO_ (, , , , primary_phandle)


   if (PROCESSHANDLE_COMPARE_ (primary_phandle,
              (short *) &receiveinfo.z_sender) != 2
           && memcmp (&receiveinfo.z_sender,
                null_phandle, PHANDLESIZE*2) != 0)

   { /*Message is not from primary process or system*/
      REPLYX ( , , , , FEOWNERSHIP);
      message.msgnumber = IGNORE_MESSAGE; }

   else

      /*Acknowledge that message was received*/
      REPLYX ( , , , , 0);

   /*Process message based on message number*/
   switch (message.msgnumber)
   {
      case CPU_DOWN:
      case PROCESS_DELETION:
         /*Backup process takes over as primary*/
```

```
            /*Start a replacement backup process*/
            initialize_backup ();

            /*Update counter with last value*/
            counter = counter_state;

            /*Continue the work of the application*/
            primary_processing ();
            break;

         case UPDATE_MESSAGE:
           /*Update the counter state*/
           counter_state =
              message.msg_variants.counter_value;
           break;
      }
   }
}
```

Function is_backup_process

Function `is_backup_process` is called by the main program to determine whether the current process is the primary process or the backup process.

```
/*Function is_backup_process returns 'true' if the
current process is the backup process and 'false'
otherwise*/
short is_backup_process (void)
{
   short process_type;

   process_type = PROCESS_GETPAIRINFO_ ();
   if (process_type == IS_BACKUP)
      return -1;  /*true*/
   return 0; /*false*/
}
```

Function main

Function `main` does the following:

- Calls `is_backup_process` to determine whether it is running as the primary or backup process.
- If it is the primary process, it initializes a backup process and then calls `primary_processing` to perform the primary processing activities.
- If it is the backup process, it calls `backup_processing` to perform the backup processing activities.

```
/*The main function determines whether it is running as the
  primary or backup process and takes appropriate
  action.*/
int main (void)
{
   if (is_backup_process ())
      backup_processing ();
   else
   {
      initialize_backup ();
      primary_processing ();
   }
}
```

Compiling and Running the Example

If creating a TNS or accelerated program, note the following before compiling, binding, and running this example:

- The `nonstoph` header is included. This header contains the declarations for the C functions that support fault-tolerant programming. (These functions are implemented in TAL. By using the `nonstoph` header, the appropriate TAL interface code is generated.)
- `#pragma search "cnonstop"` is specified. This causes the CNONSTOP object file to be bound into the program file. This object file contains the implementation of the C functions that support fault-tolerant programming.
- `#pragma wide` (for the wide-data model) is specified.

To compile the example program, enter:

```
c /in source-file/ object-file; runnable, runnamed
```

To execute the program, enter:

```
run object-file
```

If creating a TNS/R native program, note the following before compiling, linking, and running this example:

- The `crtlnsh` header is included. This header contains the declarations for the C functions that support fault-tolerant programming. (These functions are implemented in pTAL. By using the `crtlnsh` header, the appropriate pTAL interface code is generated.)
- The pragma `search "$system.system.crtlns"` is included in the compiler command line. This causes the CRTLNS object file to be bound into the program file. This object file contains the implementation of the C functions that support fault-tolerant programming.

To compile the example program, enter:

```
nmc /in source-file/ object-file; extensions, runnable, &
runnamed, search "$system.system.crtlns"
```

To execute the program, enter:

```
run object-file
```

Example With Debugging Options

The preceding example is now modified for debugging purposes. Code is added to enable the user to specify two terminal names as input parameters to the application: one for debugging the primary process and one for debugging the backup process. The example can then be run using the Inspect tool to debug the primary and backup processes from the specified terminals. Details of how this works are as follows.

When program execution is initiated by the appropriate RUND command, the Inspect tool immediately gains control of the primary process and sends a prompt to the primary process's home terminal. At this time, you can enter Inspect commands. When the command is given to resume execution, the backup process is created and started; the Inspect tool immediately gains control of the backup process and issues a prompt at the home terminal specified for the backup process. You can then enter Inspect commands to debug the backup process.

If the primary process should fail, the backup process becomes the new primary process and creates a new backup process; the home terminal for the original primary process becomes the home terminal for the new backup process, and you can continue debugging.

Following are the program modifications.

Debugging Modifications to the Program Declarations

The following variables are added to the declarations portion of the program:

```
/*Variables for debugging option*/
short arg_num;         /*Number of run-time arguments*/
char terminal_p[ZSYS_VAL_LEN_FILENAME];  /*Name of primary
                                           terminal*/
char terminal_b[ZSYS_VAL_LEN_FILENAME];   /*Name of backup
                                               terminal*/
```

The following `#include` line is added:

```
#include <cextdecs (PROCESS_GETINFOLIST_)> nolist
```

The PROCESS_GETINFOLIST_ procedure is called by the `get_terminal_name` function described later in this section.

Debugging Modifications to the Main Function

Code is added to the main function to accept two terminal names as input parameters and to save the terminal names. The new code is shown in boldface print.

```
/*The main function determines whether it is running as the
  primary or backup process and takes appropriate
  action.*/
int main (int argc, char *argv[])
{
/*Save terminal names, if provided*/
   arg_num = argc;
   if (arg_num >= 3)
   {
      strcpy (terminal_p, argv[1]);
      strcpy (terminal_b, argv[2]);
   }
   if (is_backup_process ())
      backup_processing ();
   else
   {
      initialize_backup ();
      primary_processing ();
   }
}
```

Debugging Modifications to Function initialize_backup

The name of the terminal for the backup process must be communicated to the backup process through the C function `__ns_start_backup`. Code is added to the `initialize_backup` function to test the number of parameters, get the terminal name, and pass it to `__ns_start_backup`. To get the terminal name, `initialize_backup` calls the user-written function `get_terminal_name`.

Two optional debugging parameters are added to the call to the C function `__ns_start_backup`. Note that they are intended for the PROCESS_CREATE_ procedure, which is called by `__ns_start_backup` to create the backup process.

- `terminal_name`, an input parameter that points to the string naming the home terminal for the backup process.
- `debug_options`, an input parameter that specifies debugging options. The parameter is set to 11, which indicates the following when it is passed to PROCESS_CREATE_:
  - Use the Inspect tool as the debugger.
  - Enter the Inspect tool at the first executable instruction in the main procedure.
  - If the program traps, do not create a save file.

See the *C/C++ Programmer's Guide* for details on the function parameters.

Function `initialize_backup` is as follows, with new code shown in boldface print:

```
void initialize_backup (void)
{
   short backup_phandle [PHANDLESIZE];
   char process_name [MAXNAMELEN];
   short process_name_len;
   short error;   /*value not used*/
   short error_detail;
   char *terminal_name;
   short debug_options = 11;

   /*Start the backup process*/
   if (arg_num >= 3)
   {
      /*Terminal names provided for debugging*/
      get_terminal_name (&terminal_name);
      error = __ns_start_backup (&error_detail, (short) -1,
              backup_phandle, terminal_name, debug_options);
   }
   else /*Debug option not requested*/
      error = __ns_start_backup (&error_detail, (short) -1,
                                    backup_phandle);

   /*Get the process name of the backup process*/
   error = PROCESSHANDLE_DECOMPOSE_ (backup_phandle,
                                     /*cpu*/,
                                     /*pin*/,
                                     /*nodenumber*/,
                                     /*nodename:nmax */,,
                                     /*nlen*/,
                                     process_name,
                                     MAXNAMELEN,
                                     &process_name_len);

   /*Open backup process for interprocess communication*/
   error = FILE_OPEN_ (process_name, process_name_len,
                       &backup_filenum);
}
```

Function get_terminal_name

Function `get_terminal_name` returns the home terminal name for the current backup process. It is called by `initialize_backup`. The purpose of this function is to enable you to debug multiple layers of primary process failure and backup takeover. Each time a primary process fails, the backup process becomes the new primary process and creates a new backup process. The home terminal for the new backup process is the home terminal of the previous primary process. Thus, the home terminals for the primary and backup processes are switched with each failure and takeover.

Function `get_terminal_name` does the following:

1. Calls PROCESS_GETINFOLIST_ to retrieve the name of the home terminal
2. Tests to determine which name is the correct home terminal name
3. Places the appropriate name in the return parameter

Function `get_terminal_name` is as follows:

```
void get_terminal_name (char **name)
{
   short result[40];
   short ret_attr_list[2];
   short result_length;
   short error;
   char fn[40];
   ret_attr_list[0] = 5;  /*Get home terminal name*/
   error = PROCESS_GETINFOLIST_ (,,,,, ret_attr_list, 1,
                                 result, 40, &result_length);
   strncpy (fn, (char*) &result[1], result[0]);
   fn[result[0]] = 0;
   if (strcmp (fn, terminal_b))
      *name = terminal_b;
   else
```

```
      *name = terminal_p;
}
```

Compiling and Running the Debugging Example

To compile the program for use with the Inspect tool, specify the `symbols` and `inspect` pragmas on the compiler command line.

To run the program with the Inspect tool, first locate two terminals. Then do the following:

1. On the terminal to be used for the backup process, enter PAUSE. This allows the backup process to gain control of the terminal when it begins executing.
2. On the terminal to be used for the primary process, begin execution of the program by entering:

   ```
   rund object-file $primary-terminal $backup-terminal
   ```

Active Backup Example 2

This example illustrates active backup programming for an application that performs simple I/O operations using the standard output file and files other than the standard input and standard output files. The application consists of a loop that reads a number from a file and writes the number to another file. The input and output file names are passed to the application as input parameters.

The file input to the application is a C file (type 180) with a number on each line. To create this file, first use a text editor to create an EDIT file, then convert the EDIT file to a C file with the command:

`edittoc` *`editfile`*`,` *`cfile`*

The application, without any code to make it fault-tolerant, is as follows:

```
/*This application consists of a loop that reads a number from a
file and writes it to another file*/

#pragma ansistreams   /*Create type 180 file*/
#include <stdioh> nolist
#include (stdlibh) nolist

/*File pointers*/
FILE *infile;
FILE *outfile;

int main (int argc, char *argv[])
{
   short counter;
   char sp[10];

   /*Verify at least 2 run-time arguments provided*/
   if (argc < 3)
```

```
   {
      (void) printf ("Error: Must provide input and output "
          "file names.\n");
      exit (1);
   }

   /*Open input and output files*/
   infile = fopen (argv[1], "r");
   outfile = fopen (argv[2], "ab+");


   /*Main application processing;  read number from input
   file and write it to output file and standard output*/
   for (;;)
   {
      /*Return immediately if end of file*/
      if (fgets (sp, 10, infile) == NULL) exit (0);

      counter = atoi (sp);
      printf ("counter = %d\n", counter);
      (void) fprintf (outfile, "%d\n", counter);
   }
}
```

The rest of this subsection shows the steps involved in creating an active backup version of this application.

As in the first example, the program is coded in a modular style, with individual tasks performed by separate functions. To simplify the presentation, the following conditions are not handled:

- Error conditions returned by C library functions and Guardian procedures are not handled.
- The primary process does not monitor the backup process.

Program Declarations

The particular declarations required by an active backup program depend on the application. The following are used in this example.

#include and #pragma Lines

The following `#include` and `#pragma` lines provide the declarations required by the program. The conditional statement `#ifdef _TNS_R_TARGET` causes the lines that follow it to be included only for native programs, while the lines that are between the `#else` line and the `#endif` line are included only for TNS and accelerated programs. For native programs, the pragmas `ansistreams`, `runnamed`, and `search "$system.system.crtlns"` must be included in the compiler command line.

```
#ifdef _TNS_R_TARGET
/* for native programs only */
```

```
#include <crtlnsh> nolist

/*  Note: Include the pragmas "ansistreams," "runnamed,"
*/
/*  and 'search "$system.system.crtlns"' in the compiler
*/
/*  command line for native programs.
*/

#else
/* for TNS and accelerated programs only /*
#pragma ansistreams
#pragma wide /* Use wide-data (32-bit integer) model */
#pragma runnamed /* A process pair should be named. */


#pragma search "$system.system.cnonstop"
#include <nonstoph> nolist
#endif

#include <stringh> nolist
#include <stdioh> nolist
#include <cextdecs (FILE_GETRECEIVEINFO_,\
                    FILE_OPEN_,\
                    MONITORCPUS,\
                    PROCESS_GETINFO_,\
                    PROCESS_GETPAIRINFO_,\
                    PROCESS_SETINFO_,\
                    PROCESS_STOP_,\
                    PROCESSHANDLE_COMPARE_,\
                    PROCESSHANDLE_DECOMPOSE_,\
                    READUPDATEX,\
                    REPLYX,\
                    WRITEX)> nolist
#include (stdlibh) nolist
```

`#pragma search "$system.system.cnonstop"` for TNS and accelerated programs, or `search "$system.system.crtlns"` in the compiler command line for TNS/R native programs, causes the library containing the C functions for fault-tolerant support to be bound into the program.

`#pragma wide` specifies the wide-data model (TNS and accelerated programs only).

`nonstoph` for TNS and accelerated programs, or `crtlnsh` for native programs, provides declarations required by the C functions for active backup programming. `stringh` and `stdioh` are standard header files. `cextdecs` includes declarations for the following Guardian procedures required by the program:

- FILE_GETRECEIVEINFO_, which is called by the backup process to get information about messages it receives.
- FILE_OPEN_, which is called by the primary process to open the backup process, and by the backup process to open $RECEIVE.

- MONITORCPUS, which is called by the backup process to monitor the primary process CPU. The operating system sends the backup process a “processor down” message if the monitored CPU fails.
- PROCESS_GETINFO_, PROCESS_GETPAIRINFO_, and PROCESSHANDLE_DECOMPOSE_, which are used to obtain information about the primary and backup processes.
- PROCESS_STOP_, which is called by the primary process to stop the backup process if an end-of-file is encountered in the input file.
- PROCESSHANDLE_COMPARE_, which is called by the backup process to determine if a message it received was sent by the primary process.
- READUPDATEX, which is called by the backup process to receive messages from the primary process through $RECEIVE.
- REPLYX, which is called by the backup process to reply to messages it receives.
- WRITEX, which is called by the primary process to send messages to the backup process.

Message Declarations

The program uses four message formats: two defined by the operating system and two defined by the program.

- CPU_DOWN is a system message that indicates that a monitored CPU is down. This message is used to notify the backup process if the primary process CPU fails so that the backup process can take over processing.
- PROCESS_DELETION is a system message that indicates that a process has terminated. This message is used to notify the backup process if the primary process terminates so that the backup process can take over processing.
- UPDATE_OPEN is an application-defined message that contains the open state information for the input and output files.
- UPDATE_MESSAGE is an application-defined message that contains the current state information needed by the backup process to define a continuation point and update its memory.

The message declaration portion of the program serves the following purposes:

- Provides macros that define the message numbers. Note that the system message numbers are negative and the application-defined message numbers are positive.
- Declares a structure for each of the four message formats. The first member of the structure contains the message number, and the second member is a union with one member per message format. The information contained in the system messages is not used (only the message number is used), so the exact structure is not defined.

Two other elements are defined that are used by the backup process if it receives a message it does not expect: IGNORE_MESSAGE is used as a message number and is placed in the message number portion of the message structure, and OWNERSHIP_ERROR is an error value that is returned to the sender.

The message declarations are:

```
/*message numbers*/
#define CPU_DOWN -2                 /*system message*/
#define PROCESS_DELETION -101       /*system message*/
#define UPDATE_OPEN 1               /*application message*/
#define UPDATE_MESSAGE 2            /*application message*/

#define IGNORE_MESSAGE 0            /*used by backup*/
#define OWNERSHIP_ERROR 200         /*used by backup*/

#define IS_BACKUP 6  /*Returned by PROCESS_GETPAIRINFO_ to
                     indicate whether the current process is
                     the primary or backup*/

/*The following structure defines the file open info*/
typedef struct
{
/*Input file open state*/
__ns_std_io_file_open_state input_open_info;

/*Output file open state*/
__ns_std_io_file_open_state output_open_info;
} open_info_type;

/*The following structure defines the file state
   information.*/

typedef struct
{
__ns_std_io_file_state infile_state;   /*Input file state*/
__ns_std_io_file_state outfile_state;  /*Output file state*/
__ns_std_io_file_state stdout_state;/*Standard output state*/

} update_info_type;

/*The following structure defines the various message
   formats used by this program*/

typedef struct
{
   short msgnumber; /*message number*/
   union            /*one member for each message format*/
   {
      /*UPDATE_OPEN*/
      open_info_type open_info;

      /*UPDATE_MESSAGE message*/
      update_info_type update_info;
      /*CPU_DOWN message*/
      short cpu_down_info;

      /*PROCESS_DELETION message, variable-length fields
         are assumed to have zero length*/
      short process_death_notification_info [40];
   } msg_variants;
} message_format;
```

Miscellaneous Declarations

Finally, the program contains miscellaneous declarations to define the following:

- The $RECEIVE file name.
- The maximum file-name length.
- The size of the process handle, and a null process handle. A process handle is 10 words long, and a null process handle has all ten words set to -1.
- The file number of the backup process. This number is used by the primary process to send information to the backup process.
- A variable to be used by the backup process to hold the counter state value sent by the primary process.

  The declarations are as follows:

```
#define RECEIVE_FILENAME "$RECEIVE"

/*Maximum file-name length (plus 1 to allow for null
  terminator)*/
#define MAXNAMELEN 36

/*Process handle size and null process handle*/
#define PHANDLESIZE 10
#define NULLPHANDLE {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}

/*Primary process attribute code*/
#define PRIMARY_ATTRIBUTE 47

/*File number of the backup process; used by the primary
process to send state information to the backup*/
short backup_filenum;

/*File pointers*/
FILE *infile;
FILE *outfile;
```

Creating and Starting the Backup Process

The primary process performs several activities related to initializing the backup process. Specifically, it starts the backup process and opens it for interprocess communication so that the primary process can send messages to the backup process. The operating system is allowed to select the CPU in which the backup process will run. To perform these activities, this example uses the following functions, which are the same as those used in Active Backup Example 1.

- Function `primary_cpu`, which returns the CPU number of the primary process CPU. The CPU number is required by the MONITORCPUS procedure.
- Function `initialize_backup`, which opens and starts the backup process; called by function `primary_processing`.

Updating State Information

After the primary process completes its initialization tasks, it begins executing the main processing loop. Code is added to the main processing loop to update file state information in the backup process. This code is written as a separate function, named `update_backup`, which is called by the primary process.

Function update_backup

Function `update_backup` does the following:

- Calls `backup_exists` to make sure that the backup process exists; if not, `initialize_backup` is called to recreate the backup process. See Active Backup Example 1 for the listing and description of functions `backup_exists` and `initialize_backup`.
- Calls the C function `__ns_fget_file_state` to retrieve file state information for the application input and output files and the standard output file.
- Calls WRITEX to send the file state information to the backup process.

```
/*This function is called by the primary process to update the
state of the backup process. It creates and sends a message to
the backup process.*/
void update_backup (void)
{
   short cond_code;
   short error;
   message_format message;

   /*Create update message*/
   message.msgnumber = UPDATE_MESSAGE;
   error = __ns_fget_file_state (infile,
      &message.msg_variants.update_info.infile_state);
   error = __ns_fget_file_state (outfile,
      &message.msg_variants.update_info.outfile_state);
   error = __ns_fget_file_state (stdout,
      &message.msg_variants.update_info.stdout_state);

   /*Check for existence of backup; attempt to recreate if
   necessary*/
   if (!backup_exists())
      initialize_backup();

   /*Send update message to backup*/
   cond_code = WRITEX (backup_filenum,
                       (char*) &message,
                       (short) sizeof message);
   /*Check condition code and include check for existence of
      backup*/
}
```

Primary and Backup Processing

At this stage of the example, functions have been written to perform the following active backup tasks:

- Create and start the backup process (function `initialize_backup`).
- Update state information (functions `update_backup` and `backup_exists`).

The next step is to restructure the application into separate functions for primary processing and backup processing, and a main function where execution will begin. The following new functions are created:

- Function `primary_processing`, which does the work of the application and updates state information in the backup process.
- Function `backup_processing`, which receives state information from the primary process, checks for failure of the primary process or CPU, and takes over execution if the primary process or CPU fails.
- Function `main`, which tests whether it is running as the primary process or the backup process. If it is the primary process, it creates and starts the backup

process, then calls `primary_processing`. If it is the backup process, it calls `backup_processing`.

Function primary_processing

Function `primary_processing` does the work of the application (which, in this case, is to execute a loop that reads numbers from an input file and writes the numbers to an output file and to the standard output). Within the loop, function `update_backup` is called to send file state information to the backup process. If an end-of-file is encountered in the input file, `primary_processing` stops the backup process and returns to the main function. Function `primary_processing` is as follows:

```
/*This function performs the real work of the application. It
consists of a loop that reads lines from an input file and
writes them to an output file.*/

void primary_processing (void)
{
   short counter;
   char string_pointer[10];

   for (;;)
   {
      /*Terminate immediately if end of file*/
      if (fgets (string_pointer, 10, infile) == NULL)
         exit (0); /*Stops primary and backup processes*/
      counter = atoi(string_pointer);
      (void) printf ("counter = %d\n", counter);
      (void) fprintf (outfile, "%d\n", counter);
      fflush (outfile);
      update_backup ();
   }
}
```

Function backup_processing

Function `backup_processing` handles the backup tasks, which include:

- Open $RECEIVE so that the backup process can receive messages from the primary process and the operating system.
- Call MONITORCPUS to indicate to the operating system that the backup process is to be notified if the primary process CPU fails.
- Enter an infinite loop that reads and processes messages from $RECEIVE.

If the backup process receives a message indicating that the primary process or CPU has failed, it does the following:

- Creates and starts a new backup process.
- Calls `primary_processing` to continue application processing.

The four message types received by the backup process are processed as follows:

- PROCESSOR DOWN and PROCESS DELETION indicate that the primary process has failed. The backup process takes over execution as described earlier.
- UPDATE_OPEN contains the file open state information for the input and output files. This information is used by `__ns_backup_fopen` to backup open the files.
- UPDATE_MESSAGE contains the file state information for the input and output files. This information is used by `__ns_fset_file_state` to update the file state.

Statements are included to test the process handle of the sender of the incoming message and to return an appropriate response if the sender is not the primary process or the system. If the message is from an unexpected sender, the backup replies with OWNERSHIP_ERROR. This triggers a process-handle refresh in the sender, which can then retry sending the message.

The program statements are as follows:

```
/*This function performs the backup processing: it opens
$RECEIVE, monitors the primary processor, and reads and
processes messages from $RECEIVE.*/
void backup_processing (void)
{
   short receive_filenum;
   message_format message;
   short countread;
   receive_info_type receive_info;
   short primary_value = 1;
   short cond_code;
   short error;

   /*Open $RECEIVE*/
   error = FILE_OPEN_ (RECEIVE_FILENAME,
                      (short) strlen (RECEIVE_FILENAME),
                      &receive_filenum,
                      /* access */,
                      /* exclusion */,
                      /* nowait */,
                      /* receive depth */1);
```

```
/*Monitor the primary process processor.  The backup process
receives a message if the primary process processor fails*/
MONITORCPUS ((unsigned short)(0x8000 >> primary_cpu ()));

/*Infinite loop that receives and processes messages*/
for (;;)
{
   /*Read the next message from $RECEIVE with the intention
     of replying*/

   cond_code = READUPDATEX (receive_filenum,
                            (char *) &message,
                            (short) sizeof (message_format),
                            &countread);

   /*Check process handle of sender of incoming message*/

   FILE_GETRECEIVEINFO_ ((short *) &receive_info);
   if (PROCESSHANDLE_COMPARE_ (primary_phandle,
                 (short *) &receive_info.sender_phandle) != 2
           && memcmp (&receive_info.sender_phandle,
                 null_phandle, PHANDLESIZE*2) != 0)

   { /*Message is not from primary process or system*/
      REPLYX ( , , , , OWNERSHIP_ERROR);
      message.msgnumber = IGNORE_MESSAGE; }

   else

      REPLYX ( , , , , 0);

   /*process based on message number*/
   switch (message.msgnumber )
   {
      case CPU_DOWN:
      case PROCESS_DELETION:
      /*Backup takes over*/

        /*Start replacement backup process*/
        initialize_backup ();

        /*Continue the work of the application*/
        primary_processing ();
        break;
```

```
         /*Backup open the input and output files*/
         case UPDATE_OPEN:
           infile =__ns_backup_fopen
           (&message.msg_variants.open_info.input_open_info);
           if (infile == NULL)
           {
             (void) printf ("Error in backup_processing, "
                           "update input open.\n");
             PROCESS_STOP_(,1); /*Stop primary and backup*/
            }
            outfile = __ns_backup_fopen

(&message.msg_variants.open_info.output_open_info);
            if (outfile == NULL)
            {
               (void) printf("Error in backup_processing, "
               update output open.\n");
               PROCESS_STOP_(,1); /*Stop primary and backup*/
            }
            break;

         case UPDATE_MESSAGE:
         /*Update file state information*/

            /*Update input file state*/
            error = __ns_fset_file_state (infile,
            &message.msg_variants.update_info.infile_state);

            /*Update output file state*/
            error = __ns_fset_file_state (outfile,
            &message.msg_variants.update_info.outfile_state);

            /*Update stdout state*/
            error = __ns_fset_file_state (stdout,
            &message.msg_variants.update_info.stdout_state);
            break;
      }
   }
}
```

Function main

Execution begins with the `main` function. Function `main` does the following:

- Calls `is_backup_process` to determine whether it is running as the primary process or the backup process. Function `is_backup_process` is described under Active Backup Example 1.
- If it is the backup process, calls `backup_processing` to perform the backup activities.
- If it is the primary process, does the following:
    - Opens the input and output files. The `__ns_fopen_special` function is used to open the files with a sync depth of 1.
    - Calls `initialize_backup` to initialize a backup process. Function `initialize_backup` is as described under Active Backup Example 1.
    - Calls `primary_processing` to perform the primary process activities.

Function `main` is as follows:

```
/*The main function determines whether it is executing as the
primary or backup process and takes appropriate action*/

main (int argc, char *argv[])
{
if (is_backup_process ())
   backup_processing();
else
   {
   /*Verify at least two run-time arguments provided*/
   if (argc < 3)
      {
      (void) printf ("Error: Need to provide input and "
      "output file names.\n");
      exit(1);
      }

   /*Open files with sync depth 1*/
   infile = __ns_fopen_special (argv[1], "r", 1);
   outfile = __ns_fopen_special (argv[2], "ab+", 1);
   initialize_backup ();
   primary_processing ();
   }
}
```

Compiling and Running the Example

If creating a TNS or accelerated program, note the following before compiling, binding, and running this example:

- The `nonstoph` header is included. This file contains the C library declarations for active backup programming. (This library is implemented in TAL. By using the `crtlnsh` header file, the appropriate TAL interface code is generated.)
- `#pragma search "$system.system.cnonstop"` is specified. This causes the `cnonstop` object file to be bound into the program file. This object file contains the implementation of the C functions that support fault-tolerant programming.
- `#pragma wide` (for the wide-data model) is specified.
- `#pragma ANSISTREAMS` is specified for compilation of the main function so that the input and output files are opened as ANSI files rather than EDIT files.

To compile the example program, enter:

```
c /in source-file/ object-file; runnable
```

To run the program, enter:

```
run object-file cfile output-file
```

where *`cfile`* is the C input file and *`output-file`* is the output file written by the program.

If creating a TNS/R native program, note the following before compiling, linking, and running this example:

- The `crtlnsh` header is included. This header contains the declarations for the C functions that support fault-tolerant programming. (These functions are implemented in pTAL. By using the `crtlnsh` header, the appropriate pTAL interface code is generated.)
- The pragma `search "$system.system.crtlns"` is specified in the compiler command line. This causes the CRTLNS object file to be bound into the program file. This object file contains the implementation of the C functions that support fault-tolerant programming.
- The pragma `ANSISTREAMS` is specified in the compiler command line so that the input and output files are opened as ANSI files rather than EDIT files.

To compile the example program, enter:

```
nmc /in source-file/ object-file; extensions, runnable, &
runnamed, ansistreams, search "$system.system.crtlns"
```

To execute the program, enter:

```
run object-file cfile output-file
```

where *`cfile`* is the C input file and *`output-file`* is the output file written by the program.

28 Using Floating-Point Formats

In the G07 and later versions of the NonStop operating system, users have the option of choosing between using HP floating-point format and IEEE floating-point format in their native C and C++ programs for performing floating-point arithmetic. Choosing HP floating-point format provides compatibility with pre-G07 C and C++ applications. Choosing IEEE floating-point format allows your application to take advantage of the greater performance provided by the floating-point instructions available in some CPU hardware.

HP floating-point format is supported in software millicode while IEEE floating-point format is supported in the CPU hardware. IEEE floating-point format is an industry-standard data format. HP floating-point format is HP's implementation of floating-point arithmetic. Thus, the data format is different between the two floating-point options. Data interchange with systems other than HP systems is easier with IEEE floating-point format than with Tandem floating-point format. For data interchange from one format to another, users need to call conversion routines. Tandem floating-point format is available for TNS C and C++, FORTRAN, TAL, pTAL, Pascal, COBOL85, native C and C++ programs. IEEE floating-point format is available only for native C and C++ programs.

This section discusses the following topics:

- Differences between Tandem floating-point format and IEEE floating-point format
- Building and running IEEE floating-point programs
- Compiling and linking IEEE floating-point programs
- Link-time and run-time validity checking
- Run-time support and debugging options
- Conversion and operating mode routines
- Considerations

Differences Between Tandem and IEEE Floating-Point Formats

- IEEE and Tandem floating-point data formats have different range of values and different precisions.
- IEEE and Tandem floating-point data formats have different internal layouts (for example, the exponents are in different bit fields).
- IEEE floating-point format is faster than Tandem floating-point format.
- IEEE floating-point format is easier for porting applications.
- IEEE floating-point default handling of overflow, underflow, divide-by-zero, and invalid operation is better than the Tandem floating-point handling.

- IEEE floating-point directed roundings and “sticky” flags are useful debugging tools for investigating calculations that might go wrong because of rounding problems, division by zero, or other problems. Unlike conditional flags, once sticky flags are set, they stay set until explicitly reset by the user. This capability allows the user to check a chain of computations.
- IEEE floating-point denormalized numbers avoid computational problems that arise from very small numbers as intermediate results in computations.
- IEEE floating-point format is available only on NonStop S7000 and S72000 CPUs running on NonStop operating system G07 or a later product version.
- IEEE floating-point format is the default floating-point format for TNS/E programs.

Building and Running IEEE Floating-Point Programs

If you want to build and execute programs using IEEE floating-point, make sure that the required software and hardware are available.

- IEEE floating-point format runs on S7000 and S72000 CPUs. It does not run on S70000 CPUs. For maximum application availability of IEEE floating-point programs, make sure that systems composed of S70000 and S72000 CPUs contain enough S72000 CPUs, depending on the architecture of your application.
- NonStop operating system RVU G07 or later product version.
- G06.06 or later product versions of native C and C++ compilers for compiling floating-point programs.
- G06.06 or later release of the `nld` utility for linking.

Compiling and Linking Floating-Point Programs

To use the Tandem floating-point option, set the TANDEM_FLOAT pragma (also the default) on the compiler command line. For the IEEE floating-point option, specify IEEE_FLOAT. The compiler sets the floating-point format type in the generated object file. For compiling C++ programs that use IEEE floating-point format, you must also specify the VERSION2 pragma (see the *C/C++ Programmer’s Guide* for more information).

When linking object files, if the `-set` FLOATTYPE flag is not specified, the `nld` utility derives the floating-point state from the states of the input files (see the Link-Time Consistency Checking section for more information about the `-set` FLOATTYPE flag). If one of the three floating-point types for an object file (TANDEM_FLOAT, IEEE_FLOAT, or NEUTRAL_FLOAT) is specified in the `-set` FLOATTYPE flag, `nld` sets the state of the output object file as specified by the flag.

When modifying an existing object file, `nld` sets the state as specified by the `-change` FLOATTYPE flag, which also uses one of the three floating-point types for an object file. The following is an example of compiling a program that uses IEEE floating-point format.

```
> NMC/IN SOURCEA, OUT $.#LIST/OBJECTA; IEEE_FLOAT
```

```
> NMCPLUS /IN SOURCEB, OUT $S.#LIST/OBJECTB; VERSION2, &
IEEE_FLOAT
```

In the above example, the native C compiler processes the file SOURCEA, and the native C++ compiler processes the file SOURCEB. Note that the C++ compiler has VERSION2 specified, because IEEE floating-point format is supported only under VERSION2 C++ features.

The following is an example of linking a mixed-language program that uses IEEE floating-point format.

```
> NLD $SYSTEM.SYSTEM.CRTLMAIN COBJ PTALOBJ -OBEY &
$SYSTEM.SYSTEM.LIBCOBEY -SET FLOATTYPE IEEE_FLOAT -O MYEXEC
```

In this example, the native C object file COBJ uses IEEE floating-point format, and the pTAL object file uses Tandem floating-point format. (pTAL supports only Tandem floating-point format.) To link these modules, the `-set` FLOATTYPE IEEE_FLOAT flag must be specified. If this flag is not specified, `nld` generates error messages because of the mismatch between Tandem and IEEE floating-point formats. When this flag is specified, `nld` generates a warning message about the mismatch and builds the executable file MYEXEC. CRTLMAIN (an object file) and the LIBCOBEY (an OBEY file) are standard items required when linking C programs.

The `nld` utility can use the `-set` and `-change` flags to set or change the `float_lib_overrule` attribute when creating object files. If the `float_lib_overrule` attribute is specified more than once by either the `-set` or `-change` flags, all occurrences except the last one are ignored. The `float_lib_overrule` attribute can be changed only for executable files. An error occurs if an attempt is made to change the value of this attribute for relinkable files.

Link-Time Consistency Checking

The `nld` utility checks the consistency of floating-point type combination when linking object files. The checking differs depending on whether or not the `-set` flag is specified. When the FLOATTYPE is not explicitly set with the `-set` flag, `nld` uses the FLOATTYPE attribute values of all the input object files for determining the FLOATTYPE value for the output object file. If the consistency checks of the input object files result in an invalid floating-point state or inconsistent value, an error message is generated and no output object file is created.

Any floating type combination is allowed if you explicitly override the default with the `set` FLOATTYPE flag. When the FLOATTYPE is explicitly specified with the `-set` flag, `nld` sets the FLOATTYPE value for the output object file to that specified value. A warning message and an output object file are generated if an inconsistency is detected. If the floating-point state is invalid, no output object file is created.

Execute the `noft` utility `listattribute` command to display the floating-point state. Use the `header`, `all`, `listattribute` or `filehdr` commands for displaying the `float_lib_overrule` bit.

Run-Time Consistency Checking

If you attempt to use a CPU that does not support IEEE floating-point format, process create error code 64, "IEEE floating-point support not available on this processor," occurs. The operating system does not allow the creation of a process that requires IEEE floating-point format on a CPU that does not support it. Programs can call the PROCESSOR_GETINFOLIST_ procedure to determine whether a CPU can run IEEE floating-point instructions. (See the *Guardian Procedure Calls Reference Manual* for more information about this procedure.)

The operating system also checks, at run-time, to ensure that the user program and user library have compatible floating-point types. This check can be overruled by the `float_lib_overrule` bit of the program's object file. If the `float_lib_overrule` bit is set to OFF in the program file and the user library and program have incompatible floating-point types, the operating system generates an error code; instead of running the program. (TACL converts the error code to an error message.) If the `float_lib_overrule` bit is set to ON in the program file, the operating system allows the floating-point type incompatibility between the user library and the program. See the *C/C++ Programmer's Guide* for more information about the link-time and run-time consistency checking.

Run-Time Support

One set of C and C++ run-time libraries supports both Tandem and IEEE floating-point formats. The following functions also are added to support IEEE floating-point format. For more information about these functions see the *Open System Services Library Calls Reference Manual* or the *TNS/R Native C Library Calls Reference Manual*.

- In **math.h**: `copysign()`, `isnan()`, `logb()`, `nextafter()`, and `scalb()`
- In **ieeefp.h**: `finite()`, `unordered()`, and `fpclass()`
- In **limits.h** (C++): `infinity()`, `quiet_NaN()`, `signaling_NaN()`, and `denorm_min()`

Debugging Options

You can use the Debug, Visual Inspect, and Native Inspect (TNS/E systems only) debuggers for debugging programs that use IEEE floating-point format. Note that Visual Inspect supports only the default rounding mode. Debug and Visual Inspect can be used to display and modify IEEE floating-point values and registers. Inspect can be used only for operations that do not involve IEEE floating-point values and registers.

Conversion Routines

IEEE floating-point data format is incompatible with Tandem floating-point format. Conversion between Tandem and IEEE floating-point data formats requires the use of the following routines. See the *Guardian Procedure Calls Reference Manual* for more information about these routines.

Routines for conversions from TNS to IEEE floating-point formats:

Routine	**Description**
NSK_FLOAT_TNS64_TO_IEEE64_	Converts a 64-bit TNS floating-point number to a 64-bit IEEE floating-point number.
NSK_FLOAT_TNS32_TO_IEEE64_	Converts a 32-bit TNS floating-point number to a 64-bit IEEE floating-point number.
NSK_FLOAT_TNS32_TO_IEEE32_	Converts a 32-bit TNS floating-point number to a 32-bit IEEE floating-point number.

Routines for conversions from IEEE to TNS floating-point formats:

Routine	**Description**
NSK_FLOAT_IEEE64_TO_TNS64_	Converts a 64-bit IEEE floating-point number to a 64-bit TNS floating-point number.
NSK_FLOAT_IEEE64_TO_TNS32_	Converts a 64-bit IEEE floating-point number to a 32-bit TNS floating-point number.
NSK_FLOAT_IEEE32_TO_TNS32_	Converts a 32-bit IEEE floating-point number to a 32-bit TNS floating-point number.

In addition to conversions between Tandem and IEEE floating-point formats, note that the NonStop operating system uses big-endian data formats for all data. Many other computers use little-endian data formats. For data interchanges with little-endian computers that use IEEE floating-point format, the user must reverse the order of bytes in the data.

Floating-Point Operating Mode Routines

The operating mode routines listed below are most useful for debugging programs and algorithms. These operating mode routines are used for:

- Controlling rounding mode
- Controlling denormalized number handling

- Controlling trap handling
- Controlling and accessing exception flags

Routine	**Description**
FP_IEEE_ROUND_GET_	Returns the IEEE floating-point rounding mode. FP_IEEE_ROUND_NEAREST is the default mode.
FP_IEEE_ROUND_SET_	Sets the IEEE floating-point rounding mode.
FP_IEEE_DENORM_GET_	Returns the handling of denormalized IEEE floating-point numbers. FP_IEEE_DENORMALIZATION_ ENABLE is the default mode.
FP_IEEE_DENORM_SET_	Sets the handling of denormalized IEEE floating-point numbers.
FP_IEEE_ENABLES_GET_	Returns the mask of enabled IEEE floating-point traps. By default, traps are disabled.
FP_IEEE_ENABLES_SET_	Sets the mask of enabled IEEE floating-point traps.
FP_IEEE_ENV_CLEAR_	Sets the IEEE floating-point environment to its default values and returns its previous state.
FP_IEEE_ENV_RESUME_	Sets the IEEE floating-point environment to the state returned by the FP_IEEE_ENV_CLEAR routine.
FP_IEEE_EXCEPTIONS_GET_	Returns the IEEE floating-point sticky flags (exceptions) mask. By default, no exceptions are set.
FP_IEEE_EXCEPTIONS_SET_	Sets the IEEE floating-point sticky flags (exceptions) mask.

To optimize programs, the native compiler does the following:

- Evaluates constants at compile time
- Removes unused expressions
- Reorders instructions within a procedure, but not across procedures

Because of these optimization capabilities, you might get unintended behavior if operating modes such as a special rounding mode are used. For example, setting the rounding mode does not affect constants evaluated at compile time. Further, different rounding mode settings within a procedure might be affected by instruction reordering within a procedure. When evaluating IEEE floating-point expressions, the compiler uses the default floating-point modes identified above. The expressions evaluated at compile time do not affect the exception flags.

See the *Guardian Procedure Calls Reference Manual* for details about these operating mode routines.

Operating Modes Recommendations

- Encoding rounding modes, exception flags, and other details could vary with future CPU families. Use the kfpieee.h header file for the encodings.
- Do not enable IEEE floating-point traps. To check for possible problems, clear the exception flags before a computation. Also, check the exception flags after computation. The following example illustrates the use of FP_IEEE_EXCEPTIONS_SET_ and FP_IEEE_EXCEPTIONS_GET_.

```
#include <math.h> nolist
#include <stream.h>  nolist
#include <kfpieee.h> nolist

double triangleArea /* Area of a triangle, as per Kahan */

  ( double a, double b, double c /* lengths of the sides */ )

{
   double t;
   /* sort sides so a >= b >= c */
   #define EXCHANGE(x,y) { t=x; x=y; y=t; }
   if(a<b) EXCHANGE(a,b);
   if(b<c) {
      EXCHANGE(b,c);
      if(a<b) EXCHANGE(a,b);
   }
return( sqrt( (a+(b+c))*(c-(a-b))*(c+(a-b))*(a+(b-c)) )/4 );
}  /* triangleArea */

int main (void) {
   double area;

   FP_IEEE_EXCEPTIONS_SET_( 0 );   // clear exception flags
   area = triangleArea( 1.0001, 1.0002, 2.0 );

   // test for interesting exceptions:

   if( FP_IEEE_EXCEPTIONS_GET_() &
     (FP_IEEE_INVALID|FP_IEEE_OVERFLOW|FP_IEEE_DIVBYZERO))
    {

      cout << "Trouble in computation! \n";
      return(1);
   }
    cout << "Area of the thin triangle is " << area << "\n" ;
    return(0);

}
```

Considerations

- Use 64-bit floating-point format instead of 32-bit floating-point format for greater precision, especially when doing intermediate calculations.
- Use the default operating modes for rounding and trap handling.
- IEEE floating-point values cannot be stored in SQL/MP databases.
- You cannot compile a module that uses both IEEE floating-point and embedded SQL. Compile the IEEE floating point and the embedded SQL modules separately, and then link them together.
- You cannot use IEEE floating-point values in programs that use NonStop Tuxedo software and NonStop Distributed Object Manager/MP (DOM/MP) software. These products support only Tandem floating-point values.

A Mixed Data Model Programming

The Guardian personality supports only 32-bit processes. Mixed Data Model programming is a technique through which we can create both 32-bit and 64-bit pointers for 32-bit processes.

Mixed mode programming is the model in which 32-bit programs can allocate 64-bit segments using the SEGMENT_ALLOCATE64_ procedure and access these segments using 64-bit pointers.

Note. Mixed mode programming uses 64-bit pointers to access large segments created by SEGMENT_ALLOCATE64_. In this way, 64-bit APIs can be used in the Guardian environment.

Using 64-bit Addressable Memory

Beginning with the H06.20/J06.09 RVUs, Guardian programs have access to up to 508GB of additional virtual memory. Using this additional address space can be of significant benefit to programs that need access to large amounts of in-memory data. Such programs can allocate 64-bit segments using the SEGMENT_ALLOCATE64_ procedure and then can access the segments using 64-bit pointers.

64-bit segments are allocated upward beginning at virtual address 0x100000000ULL.

Note. While NSK supports up to 508GB of virtual memory for 64-bit segments, the practical limit is determined by the amount of physical memory on the processor where the program is running, and on the Kernel Managed Swap Facility (KMSF) configuration. 64-bit segments are supported in the H06.20 and J06/09 and subsequent RVUs, but supporting system interfaces are incomplete until the H06.24 and J06.13 RVUs.

Accessing Data in 64-bit Segments

In C/C++, a 64-bit pointer is declared using the _ptr64 modifier.

For example, `char _ptr64 * longPtr`;

To use 64-bit addressing in epTAL, you must specify the __EXT64 directive on the compiler run-line or within the program source prior to the first data or procedure declaration. The epTAL compiler supporting 64-bit addressing is available in the H06.23, J06.12 and subsequent RVUs.

With __EXT64 specified, a 64-bit pointer is declared using .EXT64:

STRING .EXT64 LONGPTR;

Setting the __EXT64 option also enables the following variable types:

- EXT64ADDR - A 64-bit address
- PROC64PTR - A 64-bit pointer to a procedure

- PROC64ADDR - The 64-bit address of a procedure

Additionally, new built-ins are enabled by __EXT64:

- $EXT64ADDR_TO_EXTADDR - converts 64-bit address values to 32-bit extaddr address values; no checking is performed to see if the 32-bit address value is valid.
- $EXT64ADDR_TO_EXTADDR_OV - converts 64-bit address values to 32-bit EXTADDR address values. If the address cannot be represented in 32-bits, an overflow trap occurs.
- $EXTADDR_TO_EXT64ADDR - converts 32-bit address values to 64-bit address values.
- $FIXED0_TO_EXT64ADDR - converts a FIXED integer to a 64-bit address EXT64ADDR.
- $IS_32BIT_ADDR - Returns -1 only if the specified address value can be represented as a 32-bit byte address; otherwise, returns 0. Input values may be any of the address types except SGWADDR and SGBADDR, which are 16-bits in length.
- $PROCADDR - also accepts a PROC64ADDR expression.
- $PROC64ADDR - Converts a PROCADDR or PROC64ADDR expression to a PROC64ADDR. The bit pattern is unchanged.
- $XADR64 - 64-bit counterpart of $XADR.

Also, the $FIX function now accepts an EXT64ADDR expression, converting it to FIXED(0). The new function $UFIX converts an INT(32) into a zero-extended FIXED(0).

Allocating a 64-bit Segment

A 64-bit segment is allocated by calling SEGMENT_ALLOCATE64_(). The C/C++ prototype for the function is found in the header `kmem.h` while the epTAL external declaration is in `kmem`. The following procedures are used to allocate and manage 64-bit segments:

- SEGMENT_ALLOCATE64_

  A 64-bit version of the existing SEGMENT_ALLOCATE_ procedure.

- SEGMENT_GETINFO64_

  A 64-bit version of the SEGMENT_GETINFO_ procedure; it supports only native callers. This procedure is superseded by SEGMENT_GETINFOSTRUCT_; it is convenient for widening existing calls to SEGMENT_GETINFO_, but is not recommended for new code.

  An additional bit in the *`usage-flags`* parameter output identifies 64-bit segments. The bit is selected by mask 0x1000 (`MM_SegIs64Mask` in KMEM[.h]); in TAL notation it is bit <3>. SEGMENT_GETINFO_ assigns -1 to `base-address` for a 64-bit segment; it assigns -1 to `segment-size` if the size exceeds 31 bits.

- SEGMENT_RESIZE64_

  A 64-bit version of the existing RESIZESEGMENT procedure.

Additional procedures recognizing 64-bit segments are:

- ADDRESS_DELIMIT64_

  A 64-bit version of the existing ADDRESS_DELIMIT_ procedure.

- REFPARAM_BOUNDSCHECK64_

  A 64-bit version of the existing REFPARAM_BOUNDSCHECK_ procedure.

For more information on the SEGMENT_*64_ and related procedures, see *Guardian Procedure Calls Reference Manual*. These procedures exist in the H06.20, J06.09 and subsequent RVUs.

Dynamic Memory Allocation in 64-bit Segments

Dynamic allocation and de-allocation of space in a 64-bit segment can be accomplished through use of the 64-bit pool routines. These routines have the following advantages:

- Much higher performance than traditional NSK pool management routines
- Do not require that a pool be a contiguous block of virtual memory. A pool can be grown by allocating an additional segment, then augmenting the pool to include that segment

The 64-bit pool routines are:

- POOL64_DEFINE_ defines a new pool
- POOL64_GET_ allocates space from the pool
- POOL64_PUT_ deallocates space form the pool
- POOL64_GETINFO_ provides information about an existing pool
- POOL64_RESIZE_ grows or shrinks a single-segment pool
- POOL64_AUGMENT_ adds a potentially discontiguous memory area to a pool
- POOL64_DIMINISH_ removes a memory area previously added by POOL64_AUGMENT_
- POOL64_CHECK_SHRINK_ determines if a POOL64_RESIZE_ or POOL_DIMINISH_ call will succeed
- POOL64_CHECK_ checks the integrity of a pool

For more information on pool routines, see *Guardian Procedure Calls Reference Manual*. The POOL64_* routines are available in H06.20, J06.09 and subsequent RVUs. All TNS/E C/C++ compilers support the necessary 64-bit addressing constructs. The epTAL compiler supporting 64-bit addressing is available as of H06.23 and J06.12; the constructs must be enabled using the __EXT64 directive. The KPOOL64 header file, for use with epTAL, is available as of H06.24 and J06.13.

Data Scanning and Movement within 64-bit Segments

C/C++ programs can move data to, from and within a 64-bit segment using functions defined in `string.h`. These routines accept 64-bit pointer arguments and data lengths. Those that return a pointer, return a 64-bit pointer. The 64-bit capable functions have names that are formed from their standard C counterparts by appending '64'. For example, the 64-bit capable version of memcpy() is named memcpy64(). These functions are available in H06.22, J06.11 and subsequent RVUs.

epTAL programs can use the normal data scanning or movement constructs with 64-bit pointers.

File I/O to/from 64-bit Segments

Guardian programs can perform I/O operations directly to and from 64-bit segments using 64-bit Guardian I/O procedures. These procedures are:

- CANCELREQL
- FILE_AWAITIO64_
- FILE_COMPLETEL_
- FILE_CONTROL64_
- FILE_CONTROLBUF64_
- FILE_LOCKFILE64_
- FILE_LOCKREC64_
- FILE_READ64_
- FILE_READLOCK64_
- FILE_READUPDATE64_
- FILE_READUPDATELOCK64_
- FILE_REPLY64_
- FILE_SETMODENOWAIT64_
- FILE_UNLOCKFILE64_
- FILE_UNLOCKREC64_
- FILE_WRITE64_

- FILE_WRITEREAD64_
- FILE_WRITEUPDATE64_
- FILE_WRITEUPDATEUNLOCK64_
- FILENAME_FINDNEXT64_

Socket I/O to/from 64-bit Segments

Socket I/O can also be performed directly to and from 64-bit segments using the following Guardian Socket calls:

- send64_()
- sendto64_()
- recv64_()
- recvfrom64_()
- send_nw64_()
- send_nw2_64_()
- sendto_nw64_()
- t_sendto_nw64_()
- recv_nw64_()
- recvfrom_nw64_()
- t_recvfrom_nw64_()

OSS I/O to/from 64-bit segments

The following OSS system calls provide 64-bit I/O buffer support to 32-bit programs:

- read64_()
- recv64_()
- recvfrom64_()
- recvmsg64_()
- send64_()
- sendto64_()
- sendmsg64_()
- write64_()

Debugging Programs with 64-bit Segments

Both Visual Inspect and eInspect may be used to debug a running program that has 64-bit segments.

Snapshot files from programs that have 64-bit segments have a format that differs from those of programs without such segments. Snapshots of programs with 64-bit segments may not be examined using Visual Inspect but must rather be analyzed using eInspect.

In RVUs prior to H06.24 and J06.13:

- Only eInspect can display 64-bit segments and their contents.
- Snapshot files omit 64-bit segments; those segments simply do not exist in a debugging session using a snapshot.

Examples

Here is a complete example C program that performs the following:

- Allocates a 1GB 64-bit segment
- Defines a pool on that segment
- Moves some data into the pool
- Writes that data to stdout

C Example:

```
#include <kmem.h> nolist
#include <kpool64.h> nolist
#include <stdio.h> nolist
#include <string.h> nolist
#include <cextdecs(FILE_WRITE64_)> nolist
#include <unistd.h> nolist

int main( void ) {
   void _ptr64 * segment; /* Base address of the 64-bit Segment */
   short err, detail; /* Results from SEGMENT_ALLOCATE64_ */
   uint32 error; /* Results from POOL64 functions */
   NSK_POOL64_PTR pool_ptr; /* 64-bit Pool Header */
   void _ptr64 * ptr; /* Buffer allocated from the pool */
   int64 poolSize = 1024LL * 1024 * 1024;

   /* Allocate the 64-bit Segment */
   err = SEGMENT_ALLOCATE64_( 1
                            , poolSize
                            ,  /* filename */ , /* filename length */
                            , &detail
                            , /* pin */
                            , /* segment-type (default) */
                            , &segment
                            );
   if ( err != SEGMENT_OK ) {
      fprintf( stderr, "Error %d,%d returned by SEGMENT_ALLOCATE64_\n",
                        err, detail);
      return 1;
   }
   /* Set address of pool */
   pool_ptr = (NSK_POOL64_PTR)segment;
   /* Define the pool */
   error = POOL64_DEFINE_( pool_ptr, poolSize, POOL64Default );
   if ( error != POOL64_OK ) {
      fprintf( stderr, "Error %d returned by POOL64_DEFINE_\n" , error );
      return 1;
   }

   /* Allocate a buffer from the pool */
   ptr = POOL64_GET_( pool_ptr, 100 , &error );
   if ( error != POOL64_OK ) {
      fprintf( stderr, "Error %d returned by POOL64_GET_\n" , error );
      return 1;
   }
   /* Copy data to be written into the pool */
   strcpy64( (char _ptr64 *)ptr, "This data is in a 64-bit segment" );
   /* Write the Data to Standard Out */
   FILE_WRITE64_( STDOUT_FILENO,
                  (char _ptr64 *)ptr,
                  (int)strlen64( (char _ptr64 *)ptr )
                );

   return 0;
}
```

The following is a similar program written in epTAL:

```
?__EXT64
?SETTOG _64BIT_CALLS
?COLUMNS 79
!Global variables:
STRUCT CI_STARTUP; !Startup message
   BEGIN
   INT MSGCODE;
   STRUCT DEFAULT;
      BEGIN
      INT VOLUME[0:3];
      INT SUBVOLUME[0:3];
      END;
   STRUCT INFILE;
      BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
      END;
   STRUCT OUTFILE;
      BEGIN
      INT VOLUME[0:3];
      INT SUBVOL[0:3];
      INT FNAME[0:3];
      END;
   STRING PARAM[0:529];
   END;
INT FNO; !OUT FILE NUMBER
?NOLIST
?SOURCE KMEM( SEGMENT_PROC_CONSTANTS, SEGMENT_ALLOCATE64_ )
?SOURCE KPOOL64
?SOURCE EXTDECS( ABEND
?              , FILE_WRITE64_
?              , INITIALIZER
?              , OPEN )
?LIST

PROC START_IT(RUCB,START_DATA,MESSAGE,LENGTH,MATCH) VARIABLE;
INT .RUCB,
    .START_DATA,
    .MESSAGE,
     LENGTH,
     MATCH;
BEGIN
   CI_STARTUP.MSGCODE ':=' MESSAGE[0] FOR LENGTH/2;
END;

PROC INITIAL MAIN;
BEGIN
   INT ERR, DETAIL;
   INT(32) ERROR;
   EXT64ADDR POOLADDR;
   INT .EXT64 POOLPTR = POOLADDR;
   STRING .EXT64 PTR
        , .EXT64 PTR1;
   LITERAL POOLSIZE = 1024F * 1024F * 1024F;

   CALL INITIALIZER( !rucb!,
                     !passthru!,
                     START_IT );
   OPEN( CI_STARTUP.OUTFILE.VOLUME, FNO, %4000, 1 );
   IF <> THEN
      ABEND;
   ERR := SEGMENT_ALLOCATE64_( 1
                             , POOLSIZE
                             , ! Filename : length
                             , DETAIL
                             , ! pin
                             , ! segment_type (default)
                             , POOLADDR );
   IF ERR <> SEGMENT_OK THEN
      ABEND;
   ERROR := POOL64_DEFINE_( POOLADDR, POOLSIZE, POOL64DEFAULT );
   IF ERROR <> POOL64_OK THEN
      ABEND;
   @PTR := POOL64_GET_( POOLPTR, 100F, ERROR );
   IF ERROR <> 0d THEN
      ABEND;
   PTR ':=' "This data is in a 64-bit segment" -> @PTR1;
   FILE_WRITE64_( FNO, PTR, $DBL( @PTR1 - @PTR ) );
END;
```

Glossary

absolute pathname. An Open System Services (OSS) pathname that begins with a slash (/) character and is resolved beginning with the root directory. Contrast with relative pathname.

accelerate. To speed up emulated execution of a TNS object file by applying the Accelerator for TNS/R system execution or the Object Code Accelerator (OCA) for TNS/E system execution before running the object file.

accelerated mode. See TNS accelerated mode.

accelerated object code. The MIPS RISC instructions (in the MIPS region) that result from processing a TNS object file with the Accelerator or the Intel® Itanium® instructions (in the Itanium instruction region) that result from processing a TNS object file with the Object Code Accelerator (OCA).

accelerated object file. A TNS object file that, in addition to its TNS instructions (in the TNS region) and symbol information (in the symbol region), has been augmented by the Accelerator with equivalent but faster MIPS RISC instructions (in the MIPS region), the Object Code Accelerator (OCA) with equivalent but faster Intel® Itanium® instructions (in the Itanium instruction region), or both.

Accelerator. A program optimization tool that processes a TNS object file and produces an accelerated object file that also contains equivalent MIPS RISC instructions (called the MIPS region). TNS object code that is accelerated runs faster on TNS/R CPUs than TNS object code that is not accelerated. See also TNS Object Code Accelerator (OCA).

Accelerator region of an object file. The region (called the MIPS region) of an object file that contains MIPS RISC instructions and tables necessary to execute the object file on a TNS/R system in accelerated mode. The Accelerator creates this region. Contrast with OCA region of an object file. See also accelerated object file.

access mode. A file attribute that determines what operations you can perform on the file, like reading and writing.

active backup. A programming technique used to achieve fault tolerance in application programs. In active backup programming, a program executes as a process pair: a primary process, which performs the application processing, and a backup process, which is ready to take over execution if the primary process fails.

active process. The process that is currently using the instruction processing unit of the CPU. Contrast with inactive process.

address space. The memory locations to which a process has access.

alternate key. A sequence of characters other than the primary key used as an alternate index to records in a key-sequenced file.

alternate-key file. A key-sequenced file that provides the relationship between alternate keys and primary keys.

ancestor. The process that is notified when a named process or process pair is deleted. The ancestor is usually the process that created the named process or process pair.

ANSI. The American National Standards Institute.

APE. See application program interface (API).

API. See application program interface (API).

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 application program in the form of a client might use an API to communicate with a server program.

ASSIGN. An association of a physical file name with a logical file name made by the TACL ASSIGN command. The physical file name is any valid file name. The logical file name is used within a program. The ASSIGN is therefore used to pass file names to programs.

AXCEL. The command used to invoke the Accelerator on a TNS/R or TNS/E system.

beginning-of-tape (BOT) sticker. A light-reflecting strip that indicates the start of the usable portion of a magnetic tape. Contrast with end-of-tape (EOT) sticker.

binary semaphore. A software tool used to synchronize processes. Binary semaphores provide a way for several concurrently-executing processes to share a resource. Using binary semaphores, an executing process can hold exclusive access to the shared resource. Other processes execute until they need to use the resource, then enter a wait state until the resource becomes available.

Binder. A programming utility that combines one or more compilation units’ TNS object code files to create an executable TNS object code file for a TNS program or library. Used only with TNS object files.

Binder region. The region of a TNS object file that contains header tables for use by the Binder program.

binding. The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled TNS object files to produce a target object file.

BOT sticker. See beginning-of-tape (BOT) sticker.

BREAK mode. A mode of process execution where a process gains exclusive access to a terminal when the BREAK key is pressed. BREAK mode is established using SETPARAM function 3 or SETMODE function 11.

BREAK owner. The process that receives the Break-on-device message when the BREAK key is pressed. The establishment of BREAK ownership is achieved using SETPARAM function 3 or SETMODE function 11.

breakpoint. An object code location at which execution will be suspended so that you can interactively examine and modify the process state. With symbolic debuggers, breakpoints are usually at source line or statement boundaries.

In TNS/R or TNS/E native object code, breakpoints can be at any MIPS RISC instruction or Intel® Itanium® instruction within a statement. In a TNS object file that has not been accelerated, breakpoints can be at any TNS instruction location. In a TNS object file that has been accelerated, breakpoints can be only at certain TNS instruction locations (see memory-exact point), not at arbitrary instructions. Some source statement boundaries are not available. However, breakpoints can be placed at any instruction in the accelerated code.

C-series system. A system that is running a C-series version of the operating system.

CAID. See creator access ID (CAID).

central processing unit (CPU). Historically, the main data processing unit of a computer. HP NonStop™ servers have multiple cooperating CPUs rather than a single CPU. See also processor clock.

child process. A process created by another process. The creating process becomes the parent process of the new process. See also pathname component.

CISC compiler. See complex instruction-set computing (CISC) and TNS compiler

CISC processor. An instruction processing unit (IPU) that is based on complex instruction-set computing (CISC) architecture.

checkpoint. A line in a program at which specified information is copied from the primary process of a process pair to the backup process. This line then becomes the restart point for the backup process in the event that the primary process should stop due to CPU failure.

child process. A process created by the current process.

CISC. See complex instruction-set computing (CISC).

client. A software process, hardware device, or combination of the two that requests services from a server. Often, the client is a process residing on a programmable workstation and is the part of an application that provides the user interface. The workstation client might also perform other portions of the application logic.

client application. An application that requests a service from a shared memory. Execution of remote procedure calls is an example of a client application.

clock averaging. The technique used to keep CPU clocks synchronized.

code file. See object code file.

code segment. A segment that contains executable instructions of a program or library to be executed plus related information. Code segments can be executed and also accessed as read-only data but not written to by an application program. These read-only and execute-only segments are efficiently shared among simultaneous executions of that program or library. Therefore, they are read from disk but are never written back to disk. See also TNS code space.

code set. Codes that map a unique numeric value to each character in a character set, using a designated number of bits to represent each character. Single-byte code sets use 7 or 8 bits to represent each character. The ASCII and ISO 646 code sets use 7 bits to represent each character in Roman-based alphabets; these code sets are very limited and are not appropriate for international use. The single-byte ISO 8859 code sets use 8 bits to represent each character and can therefore support Roman-based alphabets and many others including Greek, Arabic, Hebrew, and Turkish. Multibyte code sets represent characters that require more than one byte, such as East Asian ideographic characters.

code space. that part of virtual memory reserved for user code, user library code, system code, and system library code. See TNS code space.

command-interpreter monitor ($CMON). A server process used to monitor requests made to the command interpreter (TACL process) and to affect the way the command interpreter responds.

common FCB. Seecommon file control block.

common file control block. A data structure containing information common to all SIO files opened by a process. This information includes the name of the file that receives error messages generated by SIO procedures.

Common Object File Format (COFF). A common standard for executable files and object code. On HP NonStop™ servers, COFF for TNS/R native files was replaced by the more extensible Executable and Linkable Format (ELF) beginning with the D40.00 RVU.

compiler extended-data segment. A selectable segment, with ID 1024, created and selected automatically in many (but not all) TNS processes. Within this segment, the compiler automatically allocates global and local variables and heaps that would not fit in the TNS user data segment. A programmer must keep this segment selected whenever those items might be referenced. Any alternative selections of segments must be temporary and undone before returning.

completion code. A value used to return information about a process to its ancestor process when the process is deleted. This value is returned in the Process deletion message, system message -101.

complex instruction-set computing (CISC). A CPU architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with reduced instruction-set computing (RISC).

concurrency control. The use of locking mechanisms to prevent data corruption due to concurrent access.

condition code. A status returned in the Environment Register by some file-system procedure calls to indicate whether the call was successful. A condition-code-greater-than (CCG) indicates a warning, a condition-code-less-than (CCL) indicates an error, and a condition-code-equal (=) indicates successful execution.

context-free server. A server process that does not retain any information about previous processing. It knows only about the processing of the current request message.

conversational mode. A mode of communication between a terminal and its I/O process in which each byte is transferred from the terminal to the CPU I/O buffer as it is typed. Each data-transfer operation finishes when a line-termination character is typed at the terminal. Contrast with page mode.

CPU. A functional unit of a NonStop server that is composed of one or more instruction processing units (IPUs) and memory. Each CPU has its own copy of the NonStop operating system.

***`cpu, pin`*.** In the Guardian environment, a number pair that uniquely identifies a process during the lifetime of the process, consisting of the CPU number and the process identification number (PIN).

creator. The process that causes a process to be built from a program file of another process. Compare with mom and ancestor.

creator access ID (CAID). A process attribute that identifies, by user ID, the user who initiated the process creation. Contrast with process access ID (PAID).

current selectable segment. The selectable segment that can be accessed by a process. A process specifies the current selectable segment by calling the USESEGMENT or SEGMENT_USE_ procedure to select one of a set of alternative selectable segments.

D-series system. A system that is running a D00 or later version of the operating system.

data segment. A virtual memory segment holding data. Every process begins with its own data segments for program global variables and runtime stacks (and for some libraries, instance data). Additional data segments can be dynamically created. See also flat segment and selectable segment.

data space. The area of virtual memory reserved for user data and system data.

DCT. See destination control table (DCT).

deadlock. A situation in which two processes or two transactions cannot continue because they are each waiting for the other to release a lock.

DEFINE. An HP Tandem Advanced Command Language (TACL) command you can use to specify a named set of attributes and values to pass to a process.

DEFINE name. An identifier preceded by an equal sign that can be used in place of an actual name to identify a DEFINE in a procedure call. See DEFINE.

destination control table (DCT). A collection of operating system data structures that serves as a directory of named processes and logical devices.

device subtype. A value that further qualifies a device type. For example, a device type of 4 indicates a magnetic tape drive; if the same device has a device subtype of 2, then the magnetic tape drive has a 3206 controller. You can also apply a device subtype to a process, for example, when performing terminal simulation.

disk drive. A device that stores and accesses data on a magnetic disk. Random access to addressable locations on a magnetic disk is provided by magnetic read/write heads. See also voluntary rendezvous opportunity (VRO).

disk volume. See voluntary rendezvous opportunity (VRO).

dispatching. The task of making a process active, permitting it to use the CPU to execute.

distributed system. A system that consists of a group of connected, cooperating computers.

download. The process of transferring software from one location to another, where the transferring entity initiates the transfer.

dynamic-link library (DLL). A collection of procedures whose code and data can be loaded and executed at any virtual memory address, with run-time resolution of links to and from the main program and other independent libraries. The same DLL can be used by more than one process. Each process gets its own copy of DLL static data. Contrast with shared run-time library (SRL). See also TNS/R library.

dynamic loading. Loading and opening dynamic-link libraries under programmatic control after the program is loaded and execution has begun.

duplicate key. A sequence of characters that makes up the same value in a key field in more than one record in the same file.

EDIT file. In the Guardian file system, an unstructured file with file code 101. An EDIT file can be processed by either the EDIT or PS Text Edit (TEDIT) editor. An EDIT file typically contains source program or script code, documentation, or program output. Open System Services (OSS) functions can open an EDIT file only for reading.

elapsed time. Time as measured by the CPU clock, independent of the state of any process.

`eld` utility. A utility that collects, links, and modifies code and data blocks from one or more position-independent code (PIC) object files to produce a target TNS/E native object file. See also `ld` utility and `nld` utility.

ELF. See Executable and Linkable Format (ELF).

emulate. To imitate the instruction set and address spaces of a different hardware system by means of software. Emulator software is compatible with and runs software built for the emulated system. For example, a TNS/R or TNS/E system emulates the behavior of a TNS system when executing interpreted or accelerated TNS object code.

end-of-tape (EOT) sticker. A light-reflecting strip that indicates that the end of the usable area of a magnetic tape is approaching. Contrast with beginning-of-tape (BOT) sticker.

`enoft` utility. A utility that reads and displays information from TNS/E native object files. See also `noft` utility.

EPTRACE. A performance investigation tool for Open System Services (OSS) and Guardian environments running H-series or J-series RVUs. It is designed to count, trace, locate, and provide a log of millicode-corrected misalignments in TNS stumble events.

Enscribe. A database record management system.

entry-sequenced file. A file in which each new record is stored at the end of the file in chronological sequence and whose primary key is a system-generated record address. Contrast with “key-sequenced file and relative file.

EOT sticker. See end-of-tape (EOT) sticker”

exact point. See memory manager and register-exact point.

executable. See object code file.

Executable and Linkable Format (ELF). A common standard for executable files and object code. On HP NonStop™ servers, ELF replaced the less extensible Common Object File Format (COFF) for native files.

execution mode. The emulated or real instruction set environment in which object code runs. A TNS system has only one execution mode: TNS mode using TNS compilers and 16-bit TNS instructions. A TNS/R system has three execution modes: TNS/R native mode using MIPS native compilers and MIPS instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode. A TNS/E system also has three execution modes: TNS/E native mode using TNS/E native compilers and Intel® Itanium® instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode.

exclusion mode. The attribute of a lock that determines whether any process except the lock holder can access the locked data.

executable object file. See program file.

explicit DLL. See explicit dynamic-link library (explicit DLL).

explicit dynamic-link library (explicit DLL). A dynamic-link library (DLL) that is named in the libList of a client or is a native-compiled loadfile associated with a client.

export. To offer a symbol definition for use by other loadfiles. A loadfile exports a symbol definition for use by other loadfiles that need a data item or function having that symbolic name.

EXTDECS file. See external declarations file.

extended data segment. See selectable segment.

external entry-point (XEP) table. A table located in the last page of each TNS code segment that contains links for calls (unresolved external references) out of that segment.

extensible data segment. An extended data segment for which swap file extents are not allocated until needed.

extent. A contiguous area of a disk allocated to a file.

external declarations file. A file containing external declarations for Guardian procedure calls. There are several such files relating to the high-level programming language you are using and the version of the operating system.

fault domain. In a fault-tolerant system, a module that can fail without causing a system failure.

fault tolerance. The ability of a computer system to continue processing despite the failure of any single software or hardware component within the system.

FCB. See file control block (FCB).

field. In a structured programming language, an addressable entry within a data structure.

file. An object to which data can be written or from which data can be read. A file has attributes such as access permissions and a file type. In the Open System Services (OSS) environment, file types include regular file, character special file, block special file, FIFO, and directory. In the Guardian environment, file types include disk files, processes, and subdevices.

file code. An integer value assigned to a file for application-dependent purposes, typically identifying the kind of information the file contains.

file control block (FCB). (1) A data structure automatically created and managed by the file system that contains a collection of information about a given file. (2) A data structure on the user's data stack used by SIO to access SIO files. These FCBs contain information in addition to the information kept in the FCB automatically created and managed by the file system.

file description. See open system.

file identifier. In the Guardian environment, the portion of a filename following the subvolume name. In the Open System Services (OSS) environment, a file identifier is a portion of the internal information used to identify a file in the OSS file system (an inode number). The two identifiers are not comparable.

filename. In the Open System Services (OSS) environment, a component of a pathname containing any valid characters other than slash (`/`) or null. See also file name.

file name. A string of characters that uniquely identifies a file.

In the PC environment, file names for disk files normally have at least two parts (the disk name and the file name); for example, B:MYFILE.

In the Guardian environment, disk file names include an Expand node name, volume name, subvolume name, and file identifier; for example, \NODE.$DISK.SUBVOL.MYFILE.

In the Open System Services (OSS) environment, a file is identified by a pathname; for example, `/usr/john/workfile`. See also filename.

file lock. A mechanism that restricts access to a file by all processes except the lock owner.

file name. A unique name for a file. This name is used to open a file using a FILE_OPEN_ or OPEN procedure call and thereby provides a connection between the opening process and the file.

file-name part. That portion of a file name that occurs between two periods, before the first period, or after the last period. Node name, file ID, process name, process qualifier, device name, and volume name are all examples of file-name parts.

file-name pattern. A sequence of characters including the asterisk (*) and question mark (?) that matches existing file names by expanding each asterisk to zero or more letters, digits, dollar signs ($), and pound signs (#) and replacing each question mark with exactly one letter, digit, dollar sign, or pound sign.

file-name piece. One or more consecutive parts of a file name separated by periods.

file-name subpart. An element of a file-name part separated from the next element by a colon (:).

file number. An integer that represents a particular instance of an open file. A file number is returned by the FILE_OPEN_ or OPEN procedure and is used in all subsequent input/output procedures to reference the file. Internally, the file number is an index into the file table.

file serial number. A number that uniquely identifies a file within its file system.

file system. In the Open System Services (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. Open System Services provides a file system (see also ISO/IEC IS 9945-1:1990 [ANSI/IEEE Std. 1003.1-1990], Clause 2.2.2.38); the Guardian application program interface (API) provides a file system; and OSS Network File System (NFS) provides a file system. (OSS NFS filenames and pathnames are governed by slightly different rules than OSS filenames and pathnames.) Within the OSS and OSS NFS file systems, filesets exist as manageable objects.

On an HP NonStop™ system, the Guardian file system for an Expand node is a subset of the OSS virtual file system. Traditionally, the API for file access in the Guardian environment is referred to as the *Guardian file system*.

In some UNIX and NFS implementations, the term *file system* means the same thing as *fileset*. That is, a file system is a logical grouping of files that, except for the root of the file system, can be contained only by directories within the file system. See also file system.

file type. A designation for a file structure. See entry-sequenced file, key-sequenced file. relative file, and unstructured file.

firmware. Code in memory that is necessary for the power-up initialization and communication with a host or device. The software for components of the ServerNet architecture (for example, an adapter) is called firmware. Some firmware for ServerNet components is downloaded when the system or component is loaded.

flat segment. A type of logical segment. Each flat segment has its own distinct address range within the process address space that never overlaps the range of any other allocated segments. Thus all allocated flat segments for a process are always available for use concurrently. See also logical segment and selectable segment.

fully qualified file name. The complete name of a file, including the node name. For permanent disk files, this consists of a node name, volume name, subvolume name, and file ID. For temporary disk files, consists of a node name, subvolume name and a temporary file ID. For a device, it consists of a node name and a device name or logical device number. For a named process, it consists of a node name, and a process name. For an unnamed process it consists of a node name, CPU number, PIN, and sequence number. Contrast with partially qualified file name.

global read-only array area. The part of an object file that contains global read-only arrays.

GMT. See Greenwich mean time (GMT).

godmother. Seejob ancestor.

GPLDEFS file. A file containing TAL DEFINEs used to allocate space for SIO data structures.

Greenwich mean time (GMT). The mean solar time for the meridian at Greenwich, England.

Gregorian date. A date specified according to the common calendar using the month of the year (January through December), the day of the month, and the year A.D.

graphical user interface (GUI). A user interface that offers point-and-click access to program functions.

group list. An Open System Services (OSS) process attribute that is used with the effective group ID of the process to determine the file access permissions for the process.

Guardian. An environment available for interactive or programmatic use with the HP NonStop™ operating system. Processes that run in the Guardian environment usually use the Guardian system procedure calls as their application program interface. Interactive users of the Guardian environment usually use the HP Tandem Advanced Command Language (TACL) or another HP product's command interpreter. Contrast with Open System Services (OSS).

Guardian environment. The Guardian application program interface (API), tools, and utilities.

Guardian services. An application program interface (API) to the HP NonStop™ operating system, plus the tools and utilities associated with that API. This term is synonymous with *Guardian environment*. See also Guardian.

Guardian user ID. See HP NonStop™ operating system user ID.

GUI. See graphical user interface (GUI).

header. An object that, when specified for inclusion in a program's source code, causes the program to behave as if the statement including the header were actually a specific set of other programming statements. A header contains coded information that provides details (such as data item length) about the data that the header precedes.

HP NonStop™ Open System Services (OSS). The product name for the OSS environment. See also Open System Services (OSS).

HP NonStop™ Kernel operating system. The former name of the operating system for HP NonStop systems. See HP NonStop™ operating system user ID.

HP NonStop™ operating system user ID. A user ID within an HP NonStop system. The Guardian environment normally uses the structured view of this user ID, which consists of either the *`group-number, user-number`* pair of values or the *group-name.user-name* pair of values. For example, the structured view of the super ID is (255, 255). The Open System Services (OSS) environment normally uses the scalar view of this user ID, also known as the UID, which is the value (*group-number* * 256) + *user-number*. For example, the scalar view of the super ID is (255 * 256) + 255 = 65535.

HP NonStop™ operating system. The operating system for HP NonStop systems.

HP NonStop™ System EPIC Model A processor (NSE-A processor). The model designation for the TNS/E CPU used in the HP NonStop NS10000 server.

HP NonStop™ Technical Library (NTL). The application for accessing, searching, and viewing technical publications and support information for the HP NonStop server. NTL replaces Total Information Manager (TIM).

HP Tandem Advanced Command Language (TACL). The user interface to the HP NonStop™ operating system. The TACL product is both a command interpreter and a command language. Users can write TACL programs that perform complex tasks or provide a consistent user interface across independently programmed applications.

HP Transaction Application Language (TAL). A systems programming language with many features specific to stack-oriented TNS systems.

high PIN. A PIN in the range 256 or higher.

home terminal. The terminal whose name is returned by a call to the MYTERM procedure, or the name returned in the *hometerm* parameter of the PROCESS_GETINFO_ procedure. The home terminal is often the terminal from which the process or process's ancestor was started.

hybrid shared run-time library (hybrid SRL). A shared run-time library (SRL) that has been augmented by the addition of a dynamic section that exports the SRL's symbols in a form that can be used by position independent code (PIC) clients. A hybrid SRL looks like a dynamic-link library (DLL) to PIC clients (except it cannot be loaded at other addresses and cannot itself link to DLLs). The code and data in the SRL are no different in a hybrid SRL, and its semantics for non-PIC clients are unchanged.

implicit library import library (imp-imp). See import library.

implied user library. A method of binding TNS object files that have more than 16 code segments. Segments 16 through 31 are located in the user code (UC) space but are executed as if they were segments 0 through 15 of the user library (UL) code space. This method precludes the use of a user library. Binder now supports 32 segments of UC space concurrently with 32 segments of UL code space, so the implied user library method is not needed in new or changed TNS applications.

import. To refer to a symbol definition from another loadfile. A loadfile imports a symbol definition when it needs a data item or function having that symbolic name.

import control. The characteristic of a loadfile that determines from which other loadfiles it can import symbol definitions. The programmer sets a loadfile's import control at link time. That import control can be localized, globalized, or semiglobalized. A loadfile's import control governs the way the linker and loader construct that loadfile's searchList and affects the search only for symbols required by that loadfile.

import library. A file that represents a dynamic-link library (DLL) and can substitute for it as input to the linker. Import libraries facilitate linking on auxiliary platforms (that is, PCs) where it is inconvenient to store the actual DLLs.

inactive process. A process that is not currently using the instruction processing unit of the CPU. Contrast with active process.

input/output process (IOP). A system process that drives one or more input/output units attached to the CPU through input/output channels and controllers.

interprocess communication (IPC). The exchange of messages between processes in a system or network.

Inspect region. The region of a TNS object file that contains symbol tables for all blocks compiled with the SYMBOLS directive. The Inspect region is sometimes called the SYSnn subvolume.

INSPSNAP. The program that provides a process snapshot file for the Inspect subsystem.

installation subvolume (ISV). A subvolume containing files that perform a specific function during the installation process, such as organizing documentation in a specific location, providing the components of the HP NonStop™ operating system image (OSIMAGE), and containing files that are used after the installation process.

instance. A particular case of a class of items or objects. For example, a process is defined as one instance of the execution of a program. Multiple processes might be executing the same program simultaneously. Also, instance data refers to global data of a program or library. Each process has its own instance of the data.

instance data. For each process using a dynamic-link library, a data segment area containing the global variables used by the library.

Intel® Itanium® instructions. Register-oriented Itanium instructions that are native to and directly executed by a TNS/E system. Itanium instructions do not execute on TNS and TNS/R systems. Contrast with TNS instructions and RISC instructions.

TNS Object Code Accelerator (OCA) produces Itanium instructions to accelerate TNS object code. A TNS/E native compiler produces native-compiled Itanium instructions when it compiles source code.

Intel® Itanium® instruction region. The region of a TNS object file that contains Itanium instructions and the tables necessary to execute the instructions in accelerated mode on a TNS/E system. The Object Code Accelerator (OCA) creates this region and writes it into the TNS object file. A TNS object file that contains an Itanium instruction region can execute in accelerated mode on TNS/E systems. Contrast with native.

Intel® Itanium® instruction region loading. Mapping the Itanium instructions and any tables necessary at runtime into memory from the Itanium instruction region of a TNS object file, performed when deemed necessary by the TNS emulation software on a TNS/E system.

Intel® Itanium® word. An instruction-set-defined unit of memory. An Itanium word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. Contrast with TNS word and word. See also native.

interpreted mode. See TNS interpreted mode.

interactive mode. A mode of operation that is characterized by having the same input and output device (a terminal or a process) for the session. If a terminal is used, a person enters a command and presses Return. If a process is used, the system interface waits for the process to send a request and treats the process in the same manner as a terminal. Contrast with noninteractive mode.

interrupt. The mechanism by which a CPU is notified of an asynchronous event that requires immediate processing.

interval timer. A 16-bit register that is incremented every microsecond.

IOP. See input/output process (IOP).

IPC. See interprocess communication (IPC).

IPU. The functional unit in a CPU that reads program instructions, moves data between CPU memory and the input/output controllers, and performs arithmetic operations.

job ancestor. A process that is notified when a process that is part of a job is deleted. The job ancestor of a process is the process that created the job to which the process belongs.

Julian timestamp. A 64-bit timestamp based on the Julian Date. It is a quantity equal to the number of microseconds since January 1, 4713 B.C., 12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.

Kernel subsystem. In G-series, H-series and J-series release version updates (RVUs), the subsystem for configuration and management of the Subsystem Control Facility (SCF) subsystem managers that are generic processes, some system attributes, and the ServerNet X and Y fabrics.

Kernel subsystem manager process. The graphical user interface (GUI) that starts and manages other generic processes, some system attributes, and the ServerNet X and Y fabrics in G-series release version updates (RVUs). The $ZZKRN Kernel subsystem manager process is started and managed by the $ZPM persistence manager process.

Kernel-Managed Swap Facility (KMSF). A facility by which the operating system manages virtual memory using swap files that it owns. Each CPU has at least one kernel-managed swap file that provides the swap space needs of all of its processes.

key-sequenced file. A file in which each new record is stored in sequence by primary key value, and whose primary key is either a user-defined or a system-defined value. Records in a key-sequenced file can be updated or deleted. Contrast with entry-sequenced file and relative file.

KMSF. See Kernel-Managed Swap Facility (KMSF).

keyword. A character sequence recognized by a command process.

labeled tape. A magnetic tape file described by standard ANSI or IBM file labels. Contrast with unlabeled tape.

LCT. Seelocal civil time (LCT).

LDEV. See logical device.

`ld` utility. A utility that collects, links, and modifies code and data blocks from one or more position-independent code (PIC) object files to produce a target TNS/R native object file. See also `eld` utility and `nld` utility.

legacy system. An operating system that is not open but from which applications must be ported or users transferred.

library. A generic term for a collection of routines useful in many programs. An object code library can take the form of a linkfile to be physically included into client programs, it can be an OSS archive file containing several linkable modules, it can be a loadfile, or it can be a system-managed collection of preloaded routines. Source-code libraries fall outside the scope of this glossary.

library client. A program or another library that uses routines or variables from that library.

library file. See library.

linking. The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled object files to produce a target object file.

load. (1) To transfer the HP NonStop™ operating system image or a program from disk into a computer’s memory so that the operating system or program can run.

local millicode. Emulation millicode routines that are physically copied into each TNS program code file when accelerated for TNS/E. These are a small and frequently used subset of the full set of accelerated-mode millicode routines located in the system’s

TNS Emulation Library. The program’s calls to its local copy of these routines are faster and more compact than DLL calls to the external library.

local civil time (LCT). Wall clock time in the current time zone, including any compensation for daylight saving time.

local standard time (LST). The time of day in the local time zone excluding any compensation made for daylight saving time.

logical device. (1) An addressable device, independent of its physical environment. Portions of the same logical device may be located in different physical devices, or several logical devices or parts of logical devices may be located in one physical device. (2) A process that can be accessed as if it were an I/O device; for example, the operator process is logical device LDEVOPR.

logical device number. A number that identifies a configured logical device. A logical device number can be used instead of a device file name when opening a device file.

logical memory. The portion of virtual memory that can be accessed by any process in nonprivileged mode.

logical page. (1) 2048 contiguous bytes of memory. (2) The size of a printed page given as a number of lines.

logical processor. See CPU.

logical segment. A single data area consisting of one or more consecutive 128-kilobyte unitary segments that is dynamically allocated by a process. The two types of logical segments are selectable segments and flat segments. See also selectable segment and flat segment.

login. The activity by which a user establishes a locally authenticated identity on a server. Each login has one login name.

login name. A user name associated with a session.

logon sequence. The process through which the HP NonStop™ server to be managed is determined, the security constraints to interact with that server are met, and a connection with that server is established.

low PIN. A PIN in the range 0 through 254.

LST. See local standard time (LST).

main memory. Data storage, specifically the chips that store the programs and data currently in use by a CPU.

memory-exact point. A potential breakpoint location within an accelerated object file at which the values in memory (but not necessarily the values in registers) are the same as they would be if the object file were running in TNS interpreted mode or on a TNS system. Most source statement boundaries are memory-exact points. Complex statements might contain several such points: at each function call, privileged instruction, and embedded assignment. Contrast with register-exact point and nonexact point.

memory manager. An HP NonStop™ operating system process that implements the paging scheme for virtual memory. This process services requests generated by different interrupt handlers as well as by other system processes.

memory page. A unit of virtual storage. In TNS systems, a memory page contains 2048 bytes. In TNS/R systems, the page size is determined by the memory manager and can vary, depending on the CPU type.

memory pool. An area of an extended data segment or user data segment that your program allocates and from which your program can obtain and release blocks of storage.

message system. A set of operating system procedures and data structures that handles the mechanics of exchanging messages between processes.

message tag. A value assigned to a message by the file system when you read the message from $RECEIVE. You can use this tag to identify the reply when processing multiple messages at the same time.

MIPS Computer Systems, Incorporated. RISC CPU manufacturer.

MIPS region of a TNS object file. The region of a TNS object file that contains MIPS instructions and the tables necessary to execute the instructions in accelerator mode on a TNS/R system. Accelerator creates this region and writes it into the TNS object file. Contrast with Intel® Itanium® instruction region.

MIPS RISC instructions. Register-oriented 32-bit machine instructions in the MIPS-1 RISC instruction set that are native to and directly executed on TNS/R systems. MIPS RISC instructions do not execute on TNS systems and TNS/E systems. Contrast with TNS instructions and Intel® Itanium® instructions.

Accelerator-generated MIPS RISC instructions are produced by accelerating TNS object code. Native-compiled MIPS RISC instructions are produced by compiling source code with a TNS/R native compiler.

MIPS RISC word. An instruction-set-defined unit of memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. Contrast with TNS word and word. See also Intel® Itanium® word.

millicode. RISC instructions that implement various TNS low-level functions such as exception handling, real-time translation routines, and library routines that implement the TNS instruction set. Millicode is functionally equivalent to TNS microcode.

mirrored disk. A pair of identical disk drives that are used together as a single volume. One drive is considered the primary and the other is called the backup or the mirror. Each byte of data written to the primary drive is also written to the backup drive; if the primary drive fails, the backup can continue operations.

mom. A process that is notified when certain other processes are deleted. In a process pair, each process is the other's mom When a process is unnamed, its mom is usually the process that created it.

monitor. A process that, among other functions, is responsible for checking that certain other processes continue to run. If such a process should stop, then it is the monitor's responsibility to restart it.

multibyte character set. A means for identifying written characters for national languages that require more than one byte to represent a single character.

multiprocessing. Two or more processes running in parallel by executing on different processing modules.

multithreaded process. A process that performs more than one operation concurrently. Contrast with single-threaded process.

named process. A process to which a process name was assigned when the process was created. Contrast with unnamed process.

native. An adjective that can modify object code, object file, process, procedure, and mode of process execution. Native object files contain native object code, which directly uses MIPS or Intel® Itanium® instructions and the corresponding conventions for register handling and procedure calls. Native processes are those created by executing native object files. Native procedures are units of native object code. Native mode execution is the state of the process when it is executing native procedures.

native mode. See TNS/R native mode and TNS/E native mode.

native-mode code. Object code that has been compiled with TNS/R native compilers to run on TNS/R systems or with TNS/E native compilers to run on TNS/E systems.

native-mode library. A native-compiled loadfile associated with one or more other native-compiled loadfiles. A native mode process can have any number of associated native-mode libraries. See also TNS library and TNS/E library.

native-mode source code. High-level language routines that can be compiled with TNS/E native compilers. These two sets of compilers accept the same language dialects.

native object code. See TNS/R native object code and TNS/E native object code.

native object file. See TNS/R native object file and TNS/E native object file.

native process. SeeTNS/R native process and TNS/E native process.

native signal. SeeTNS/R native signal and TNS/E native signal.

`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. See also `eld` utility and `ld` utility.

node. A system of one or more CPU modules. Typically, a node is linked with other nodes to form a network.

node name. The portion of a file name that identifies the system through which the file can be accessed.

`noft` utility. A utility that reads and displays information from TNS/R native object files. See also `enoft` utility.

nonexact point. A code location within an accelerated object file that is between memory-exact points. The mapping between the TNS program counter and corresponding RISC instructions is only approximate at nonexact points, and interim changes to memory might have been completed out of order. Breakpoints cannot be applied at nonexact points. Contrast with memory manager and register-exact point.

nonretryable error. An error condition returned by the file system that cannot be recovered by retrying the operation even after operator intervention. Contrast with retryable error.

NonStop™ Kernel operating system. See HP NonStop™ operating system.

NonStop™ Series TNS. See temporary disk file.

NonStop™ Series TNS/E. See TNS/E.

NonStop™ Series TNS/R. See TNSVU.

NonStop Technical Library. See HP NonStop™ Technical Library (NTL).

NonStop Kernel Open System Services (OSS). An application program interface (API) to the Tandem NonStop Kernel and associated tools and utilities. SeeOpen System Services (OSS) for a more complete definition.

NonStop SQL. A relational database management system that provides efficient online access to large distributed databases.

NSAA. HP NonStop™ advanced architecture.

NSE-A processor. See HP NonStop™ System EPIC Model A processor (NSE-A processor).

`noft` utility. A utility that reads and displays information from TNS/R native object files. See also `enoft` utility.

noninteractive mode. A mode of operation that usually involves a command file (an EDIT file that contains a series of commands). Contrast with interactive mode.

nowait I/O. An operation with an I/O device or process where the issuing process does not wait for the I/O operation to finish. Contrast with waited I/O.

object code accelerator (OCA). See TNS Object Code Accelerator (OCA).

object code file. A file containing compiled machine instructions for one or more routines. This file can be an executable loadfile for a program or library or a not-yet-executable linkfile for some program module. On other systems, an object code file is also known as a *binary* or as an *executable*.

object code interpreter (OCI). See TNS Object Code Interpreter (OCI).

OCA. (1) The command used to invoke the TNS Object Code Accelerator (OCA) on a TNS/E system. (2) See TNS Object Code Accelerator (OCA).

OCA region loading. A task performed when necessary by the TNS emulation software for TNS/E machines. This task involves mapping into memory the Intel® Itanium® instructions and any tables needed at run time from the TNS Object Code Accelerator (OCA) region (called the Itanium instruction region) of an OCA-generated object file.

OCA region of an object file. The region of a TNS Object Code Accelerator (OCA)-generated object file, also called the Intel® Itanium® instruction region, that contains Itanium instructions and tables necessary to execute the object file on a TNS/E system in TNS accelerated mode. The TNS Object Code Accelerator (OCA) creates this region. See also OCA-accelerated object code. Contrast with Accelerator region of an object file.

OCA-accelerated object code. The Intel® Itanium® instructions that result from processing a TNS object file with the TNS Object Code Accelerator (OCA).

OCA-accelerated object file. A TNS object file that has been augmented by the TNS Object Code Accelerator (OCA) with equivalent but faster Intel® Itanium® instructions. An OCA-accelerated object file contains the original TNS object code, the OCA-accelerated object code and related address map tables, and any Binder and symbol information from the original TNS object file. An OCA-accelerated object file also can be augmented by the Accelerator with equivalent MIPS RISC instructions.

OCA-generated Itanium instructions. See Intel® Itanium® instructions.

OCI. See TNS Object Code Interpreter (OCI).

object code library. Synonym for library.

object file. A file generated by a compiler or linker that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file may be a complete program that is ready for immediate execution, or it may be incomplete and require linking with other object files before execution. See also TNS object file, TNS/R native object file, and TNS/E native object file.

one-way communication. A form of interprocess communication where the sender of a message (the requester) does not expect any data in the reply from the receiver of the message (the server). Contrast with two-way communication.

open system. A system with interfaces that conform to international computing standards and therefore appear the same regardless of the system's manufacturer. For example, the Open System Services (OSS) environment on HP NonStop™ systems conforms to international standards such as ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990, also known as POSIX.1), national standards such as FIPS 151-2, and portions of industry specifications such as the X/Open Portability Guide Version 4 (XPG4).

Open System Services (OSS). An open system environment available for interactive or programmatic use with the HP NonStop™ operating system. Processes that run in the OSS environment usually use the OSS application program interface. Interactive users of the OSS environment usually use the OSS shell for their command interpreter. Synonymous with *Open System Services (OSS) environment*. Contrast with Guardian.

Open System Services (OSS) environment. The HP NonStop™ Open System Services (OSS) application program interface (API), tools, and utilities.

opener table. A data structure maintained by a server process containing information about processes that have the server process open. Typically, the server uses this table to check that incoming requests originate from valid openers.

operating system. See HP NonStop™ operating system.

operational environment. The conditions under which your system performs. These conditions include the devices and communications lines that are made active and the system and application processes that are started at system startup.

OS. See HP NonStop™ operating system.

OSS. See Open System Services (OSS).

OSS environment. See Open System Services (OSS) environment.

OSS process ID. In the Open System Services (OSS) environment, the unique identifier that identifies a process during the lifetime of the process and during the lifetime of the process group of that process.

OSS signal. A signal model defined in the POSIX.1 specification and available to TNS processes and TNS/R native processes in the OSS environment. OSS signals can be sent between processes.

OSS user ID. See HP NonStop™ operating system user ID.

page. A unit of virtual storage for disks and CPU memory. The size of a disk page is 2048 bytes. The size of a memory page varies depending on the CPU type and the software release.

page mode. A mode of communication between a terminal and its I/O process in which the terminal stores up to a full page of data (1920 bytes) in its own memory before sending the page to the I/O process. Contrast with conversational mode.

PAID. See process access ID (PAID).

PARAM. An association of an ASCII value with a parameter name made by the TACL PARAM command. You can use PARAMs to pass parameter values to processes.

partially qualified file name. A file name in which only the right-hand file-name parts are specified. The remaining parts of the file name assume default values. Contrast with fully qualified file name.

pathname. In the Open System Services (OSS) file system and Network File System (NFS), the string of characters that uniquely identifies a file within its file system. A pathname can be either relative or absolute. See also ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990 or POSIX.1), Clause 2.2.2.57.

pathname component. See filename.

partitioned file. A logical file made up of several partitions that can reside on different disks. Generic key values determine the partition on which a given record resides.

PCB. See process control block (PCB).

permanent disk file. A file that remains on disk until it is explicitly purged.

persistent process. A process that must always be either waiting, ready, or executing. These processes are usually controlled by a monitor process that checks on the status of persistent processes and restarts them if necessary.

PFS. See process file segment (PFS).

physical memory. The semiconductor memory that is part of every CPU.

PIN. See process identification number (PIN).

primary extent. The first contiguous area of disk allocated to a file. See also secondary extent.

primary key. A unique value that identifies a record in a structured disk file.

primary path. The path between the CPU and controller that is currently being used.

priority. An indication of the precedence with which a process gains access to the instruction processing unit.

process. A program that has been submitted to the operating system for execution. Multiple submissions of the same program run multiple processes.

process access ID (PAID). A user ID used to determine whether a process can make requests to the system, for example to open a file, stop another process, and so on. The process access ID is usually the same as the creator access ID but it can be different; the owner of the corresponding object file can set the object file security such that it runs with a process access ID equal to the user ID of the file owner, rather than the creator of the process. Contrast withcreator access ID (CAID).

process control block (PCB). An operating system data structure that contains information about the resources and environment of a process.

process descriptor. A process name returned by a Guardian procedure call.

process environment. The state and contents of the code and data spaces, stacks, and register values that exist when the IPU is executing instructions that are part of a user or system process.

process file name. A file name that identifies a process.

process file segment (PFS). An extended data segment that is automatically allocated to every process and contains operating system data structures, file-system data structures, and memory-management pool data structures.

process handle. A D-series 20-byte data structure that identifies a named or unnamed process in the network. A process handle identifies an individual process; thus, each process of a process pair has a unique process handle.

process ID. In the Guardian environment, the content of a 4-integer array that uniquely identifies a process during the lifetime of the process.

process identification number (PIN). A number that uniquely identifies a process running in a CPU. The same number can exist in other CPUs in the same system. See also process ID.

process name. A name that can be assigned to a process when the process is created. A process name uniquely identifies a process or process pair in a system.

process pair. Two processes created from the same object file running in a way that makes one process a backup process of the other. Periodic checkpointing ensures that the backup process is always ready to take over from the primary if the primary process should fail. The process pair has one process name but each process has a different process identification number (PIN).

process qualifier. A suffix to a process file name that gets passed to a process when the process is opened. Its use is application-dependent.

process sequence number. A subpart of a process file name that allows the process to be identified over time.

process time. The amount of time that a process has spent in the active substate.

process timer. A clock that measures process execution time.

processor clock. A hardware timer on each CPU module that keeps CPU time; the number of microseconds since cold load.

processor time. The time represented by a CPU clock.

program. See program file.

program file. An executable object code file containing a program's main routine plus related routines statically linked together and combined into the same object file. Other routines shared with other programs might be located in separately loaded libraries. A program file can be named on a RUN command; other code files cannot. See also object code file.

ready list. A linked list of PCBs of ready processes.

ready process. A process that is prepared to become active.

read-only data segment. An extended data segment, the contents of which a process can read but cannot alter. The contents of a read-only data segment can be shared across CPUs.

record block. A collection of data records written to or read from a mass storage medium using one I/O operation. Record blocks are usually used with magnetic tape to speed I/O.

record lock. A lock held by a process or a transaction that restricts access to that record by other processes.

reduced instruction-set computing (RISC). A CPU architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with complex instruction-set computing (CISC).

register-exact point. A synchronization location within an accelerated object file at which both of these statements are true:

All live TNS registers plus all values in memory are the same as they would be if the object file were running in TNS mode or TNS interpreted mode or on a TNS system.

All accelerator code optimizations are ended.

Register-exact points are a small subset of all memory-exact points. Procedure entry and exit locations and call-return sites are usually register-exact points. All places where the program might switch into or from TNS mode or TNS interpreted mode are register-exact points. Contrast with memory manager and nonexact point.

relative file. A file in which each new record is stored at the relative record location specified by its primary key, and whose primary key is a relative record number. Records can be updated or deleted. Contrast with key-sequenced file and entry-sequenced file.

relative pathname. In the Open System Services (OSS) file system and Network File System (NFS), a pathname that does not begin with a slash (/) character. A relative pathname is resolved beginning with the current working directory. Contrast with absolute pathname.

release version update (RVU). A collection of compatible revisions of HP NonStop operating system software products, identified by an RVU ID, and shipped and supported as a unit. An RVU consists of the object modules, supporting files, and documentation for the product revisions. An RVU also includes a set of documentation for the overall RVU.

rendezvous. A distributed operation that exchanges data to achieve synchronization among the CPU elements of a logical processor.

RVU. See release version update (RVU).

reply. A response to a requester process by a server process. Contrast with request.

request. A message formatted and sent to a server by a requester. Requests also include status messages such as CPU up and CPU down messages, which are placed on the intended recipient’s process message queue ($RECEIVE file) by the operating system. Contrast with reply.

requester. A process that initiates interprocess communication by sending a request to another process. Contrast with server.

response. See reply.

retryable error. An error condition returned by the file system that can be corrected by repeating the operation that caused the error. Sometimes operator intervention is required before the retry, for example to put paper into an empty printer. Contrast with nonretryable error.

RISC. See reduced instruction-set computing (RISC)**.**

RISC instructions. Register-oriented 32-bit machine instructions that are directly executed on TNS/R CPUs. RISC instructions execute only on TNS/R systems, not on TNS systems. Contrast with TNS instructions.

RUCB. See “run unit control block (RUCB).”

run unit control block (RUCB). A data structure used by the INITIALIZER procedure to specify to SIO the name of the common FCB and the number other SIO FCBs to be created.

secondary extent. A contiguous area of disk storage allocated to a file. A file is made up of one or more extents; the first extent is the primary extent, and other extents are secondary extents. The secondary extents are all the same size for a specific file. The primary extent can be a different size. See also primary extent."

segment. In general, a contiguous sequence of logically related pages of virtual memory. The pages of the segment are individually swapped in and out of physical memory as needed. Within a loadable object file, one of the portions of the file that is mapped as one unit into virtual memory as the file is loaded. See also code segment and data segment.

segment ID. A number that identifies an extended data segment and specifies the kind of extended data segment to allocate.

selectable segment. A type of logical segment formerly known as an extended data segment. The data area for a selectable segment always begins with relative segment 4, and this area can be dynamically switched among several selectable segments by calls to the Guardian SEGMENT_USE_ procedure. The effect is similar to a rapid overlaying of one large data area. See also logical segment and flat segment.

sequence number. See process sequence number.

server. (1) An implementation of a system used as a stand-alone system or as a node in an Expand network. (2) A combination of hardware and software designed to provide services in response to requests received from clients across a network. For example, HP NonStop™ servers provide transaction processing, database access, and other services. (3) 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.

shared data segment. An extended data segment that can be accessed by more than one process.

shared memory. An interprocess communication mechanism that allows two or more processes to share a given region of memory.

Shared Millicode Library. An intrinsic library containing privileged or TNS-derived millicode routines used by many native-compiled programs and by emulated TNS programs. This library includes efficient string-move operations, TNS floating-point emulation, and various privileged-only operations. These routines are mode independent. They comply with native calling conventions but can be directly invoked from any mode without changing execution modes.

shared run-time library (SRL). An object file that the operating system links to a program file at run time. See also TNS/R native shared run-time library (TNS/R native SRL).

signal. A means by which a process can be notified of or affected by an event occurring in the system. Some signals are used to notify a process when an error not related to input or output has occurred. See also TNS/R native signal and OSS signal. Contrast with trap.

signal handler. A procedure that is executed when a signal is received by a process.

single-threaded process. A process that performs only one operation at a time. Contrast with multithreaded process.

site update tape (SUT). One or more tapes that contain each target system's site-specific subvolume and various products. Each product contains a softdoc and a complete set of files. A SUT is delivered with every new HP NonStop™ system and can be ordered whenever a new release version update (RVU) of the system software is available. A full SUT contains the current RVU of the HP NonStop operating system and all product software that has been ordered with it. A partial SUT contains a subset of products for the current RVU.

SRL. See shared run-time library (SRL).

startup sequence. A convention for sending and receiving certain messages while starting a new process. By convention, the new process receives an Open message, followed by a Startup message, an Assign message for each ASSIGN in effect, a Param message if there are any PARAMs in effect, and then a Close message.

state information. In active backup programming, information about a program's execution environment that is periodically sent from the primary process to the backup process. In the event of a primary process failure, the backup process uses the state information to take over execution at or near the logical point of failure.

stop mode. An attribute that a process can use to protect itself from being stopped by other processes.

structured file. A disk file in which records are saved according to a predefined structure. See relative file. key-sequenced file, and entry-sequenced file for examples. Contrast with unstructured file.

subvolume. A group of files stored on disk. These files all have the same subvolume name but each has a different file ID. For example, $DATA.INFO identifies the subvolume named INFO on the volume $DATA. An example of a file name is $DATA.INFO.RESULTS.

super ID. On HP NonStop™ systems, a privileged user who can read, write, execute, and purge all files on the system. The super ID is usually a member of a system-supervisor group.

The super ID has the set of special permissions called appropriate privileges. In the Guardian environment, the structured view of the super ID, which is (255, 255), is most commonly used. In the Open System Services (OSS) environment, the scalar view of the super ID, which is 65535, is most commonly used.

superuser. See super ID.

SUT. See site update tape (SUT).

swap files. The disk files to and from which information is copied during swapping. The Kernel-Managed Swap Facility (KMSF) now manages swap space for most of the purposes for which processes formerly maintained separate swap files. See Kernel-Managed Swap Facility (KMSF).

swapping. The process of copying information between physical memory and disk storage.

sync ID. A value used by the operating system 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.

SYS*nn* subvolume. A subvolume on the $SYSTEM volume where the new version of the HP NonStop™ operating system image is located. Also located on the SYS*nn* subvolume is system-dependent and release version update (RVU)-dependent software. *nn* is an octal number in the range %00 through %77.

system. All the CPUs, controllers, firmware, peripheral devices, software, and related components that are directly connected together to form an entity that is managed by one HP NonStop™ operating system image and operated as one computer.

System Code (SC). See TNS code space.

system code. A logically distinct part of the HP NonStop™ operating system that consists of operating system procedures shared by all CPUs.

system entry-point (SEP) table. A table used on TNS systems that stores the XEP entry value for each TNS operating system procedure entry point.

system expansion. The process of making a target system larger by adding enclosures to it. The enclosures being added can be either new enclosures or enclosures from a donor system. Contrast with TACL.

system generation. The process of creating an operating system to support a particular system configuration and release version update (RVU).

System Library (SL). See TNS code space.

system library. A logically distinct part of the HP NonStop™ operating system that consists of user-callable library procedures and kernel procedures.

system process. (1) A privileged process that comes into existence at system-load time and exists continuously for a given configuration for as long as the CPU remains operable. (2) An HP NonStop™ operating system process, such as the memory manager, the monitor, and the input/output (I/O) control processes. The files containing system processes are invoked by ALLPROCESSORS paragraph entries. (3) A part of a single copy of the HP NonStop operating system with Open System Services (OSS) interfaces. A system process does not have an OSS process ID.

system time. The time represented by any synchronized CPU clock in the system.

TACL. See HP Tandem Advanced Command Language (TACL).

TAL. See HP Transaction Application Language (TAL).

temporary disk file. A file stored on disk that will be purged automatically as soon as the process that created it stops.

terminal-simulation process. A process that is made to behave like a terminal file.

TIM. See HP NonStop™ Technical Library (NTL).

timekeeping. A function performed by the operating system that involves initializing and maintaining the correct time in a CPU.

timestamp. An item containing a representation of the time. A timestamp can be applied to an object at some critical point in the object's life.

TNS. Refers to fault-tolerant HP computers that support the HP NonStop™ operating system and are based on microcoded complex instruction-set computing (CISC) technology. TNS systems run the TNS instruction set. Contrast with TNS/R and TNS/E.

TNS accelerated mode. A TNS emulation environment on a TNS/R or TNS/E system in which accelerated TNS object files are run. TNS instructions have been previously translated into optimized sequences of MIPS or Intel® Itanium® instructions. TNS accelerated mode runs much faster than TNS interpreted mode. Accelerated or interpreted TNS object code cannot be mixed with or called by native mode object code. See also TNS Object Code Accelerator (OCA). Contrast with TNS/R native mode and TNS/E native mode.

TNS C compiler. The C compiler that generates TNS object files. Contrast with TNS/E native compiler.

TNS code segment. One of up to 32 128-kilobyte areas of TNS object code within a TNS code space. Each segment contains the TNS instructions for up to 510 complete routines.

TNS code segment identifier. A seven-bit value in which the most significant two bits encode a code space (user code, user library, system code, or system library) and the five remaining bits encode a code segment index in the range 0 through 31.

TNS code segment index. A value in the range 0 through 31 that indexes a code segment within the current user code, user library, system code, or system library space. This value can be encoded in five bits.

TNS code space. One of four addressable collections of TNS object code in a TNS process. They are User Code (UC), User Library (UL), System Code (SC), and System Library (SL). UC and UL exist on a per-process basis. SC and SL exist on a per-node basis.

TNS compiler. A compiler in the TNS development environment that generates 16-bit TNS object code following the TNS conventions for memory, stacks, 16-bit registers, and call linkage. The TNS C compiler is an example of such a compiler. Contrast with TNS/E native compiler.

TNS Emulation Library. A public dynamic-link library (DLL) on a TNS/E system containing the TNS Object Code Interpreter (OCI), millicode routines used only by accelerated mode, and millicode for switching among interpreted, accelerated, and native execution modes. See also Shared Millicode Library.

TNS emulation software. The set of tools, libraries, and system services for running TNS object code on TNS/E systems and TNS/R systems. On a TNS/E system, the TNS emulation software includes the TNS Object Code Interpreter (OCI), the TNS Object Code Accelerator (OCA), and various millicode libraries. On a TNS/R system, the TNS emulation software includes the TNS Object Code Interpreter (OCI), the Accelerator, and various millicode libraries.

TNS instructions. Stack-oriented, 16-bit machine instructions that are directly executed on TNS systems by hardware and microcode. TNS instructions can be emulated on TNS/E and TNS/R systems by using millicode, an interpreter, and either translation or acceleration. Contrast with reduced instruction-set computing (RISC) and Intel® Itanium® instructions.

TNS interpreted mode. A TNS emulation environment on a TNS/E system in which individual TNS instructions in a TNS object file are directly executed by interpretation rather than permanently translated Intel® Itanium® instructions. TNS interpreted mode runs slower than TNS accelerated mode. Each TNS instruction is decoded each time it is executed, and no optimizations between TNS instructions are possible. TNS interpreted mode is used when a TNS object file has not been accelerated for that hardware system, and it is also sometimes used for brief periods within accelerated object files. Accelerated or interpreted TNS object code cannot be mixed with or called by native mode object code. See also TNS Object Code Interpreter (OCI). Contrast with TNS accelerated mode, TNS/R native mode, and TNS/E native mode.

TNS library. A single, optional, TNS-compiled loadfile associated with one or more application loadfiles. If a user library has its own global or static variables, it is called a TNS shared run-time library (TNS SRL). Otherwise it is called a User Library (UL).

TNS mode. The operational environment in which TNS instructions execute by inline interpretation. See also accelerated mode.

TNS object code. The TNS instructions that result from processing program source code with a TNS compiler. TNS object code executes on TNS, TNS/R, and TNS/E systems.

TNS Object Code Accelerator (OCA). A program optimization tool that processes a TNS object file and produces an accelerated file for a TNS/E system. OCA augments a TNS object file with equivalent Intel® Itanium® instructions. TNS object code that is accelerated runs faster on TNS/E systems than TNS object code that is not accelerated. See also Accelerator and TNS Object Code Interpreter (OCI).

TNS Object Code Interpreter (OCI). A program that processes a TNS object file and emulates TNS instructions on a TNS/E system without preprocessing the object file. See also TNS Object Code Accelerator (OCA).

TNS object file. An object file created by a TNS compiler or the Binder. A TNS object file contains TNS instructions. TNS object files can be processed by the Accelerator or by the TNS Object Code Accelerator (OCA) to produce to produce accelerated object files. A TNS object file can be run on TNS, TNS/R, and TNS/E systems.

TNS procedure label. A 16-bit identifier for an internal or external procedure used by the TNS object code of a TNS process. The most-significant 7 bits are a TNS code segment identifier: 2 bits for the TNS code space and 5 bits for the TNS code segment index. The least-significant 9 bits are an index into the target segment's procedure entry-point (PEP) table. On a TNS/E system, all shells for calling native library procedures are segregated within the system code space. When the TNS code space bits of a TNS procedure label are %B10, the remaining 14 bits are an index into the system's shell map table, not a segment index and PEP index.

TNS process. A process whose main program object file is a TNS object file, compiled using a TNS compiler. A TNS process executes in interpreted or accelerated mode while within itself, when calling a user library, or when calling into TNS system libraries. A TNS process temporarily executes in native mode when calling into native-compiled parts of the system library. Object files within a TNS process might be accelerated or not, with automatic switching between accelerated and interpreted modes on calls and returns between those parts. Contrast with TNS/R native process and TNS/E native process.

TNS stack segment. See TNS user data segment.

TNS State Library for TNS/E. A library of routines to access and modify the TNS state of a TNS process running on TNS/E.

TNS system library. A collection of HP-supplied TNS-compiled routines available to all TNS processes. There is no per-program or per-process customization of this library. All routines are immediately available to a new process. No dynamic loading of code or creation of instance data segments is involved. See also HP NonStop™ operating system.

TNS user data segment. In a TNS process, the segment at virtual address zero. Its length is limited to 128 kilobytes. A TNS program's global variables, stack, and 16-bit heap must fit within the first 64 kilobytes. See also compiler extended-data segment.

TNS user library. A user library available to TNS processes in the Guardian environment.

TNS word. An instruction-set-defined unit of memory. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. See also Intel® Itanium® word, native, and word.

TNS/E. Refers to fault-tolerant HP computers that support the HP NonStop™ operating system and are based on the Intel® Itanium® CPU. TNS/E systems run the Itanium instruction set and can run TNS object files by interpretation or after acceleration. TNS/E systems include all HP NonStop™ systems that use NSE-*x* CPUs. Contrast with temporary disk file.

TNS/E library. A TNS/E native-mode library. TNS/E libraries are always dynamic-link libraries (DLLs); there is no native shared runtime library (SRL) format.

TNS/E native compiler. A compiler in the TNS/E development environment that generates TNS/E native object code, following the TNS/E native-mode conventions for memory, stack, registers, and call linkage. The TNS/E native C compiler is an example of such a compiler. Contrast with TNS compiler.

TNS/E native mode. The primary execution environment on a TNS/E system, in which native-compiled Intel® Itanium® object code executes, following TNS/E native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage. Contrast with TNS interpreted mode and TNS accelerated mode.

TNS/E native object code. The Intel®® Itanium® instructions that result from processing program source code with a TNS/E native compiler. TNS/E native object code executes only on TNS/E systems, not on TNS systems or TNS/R systems.

TNS/E native object file. An object file created by a TNS/E native compiler that contains Intel® Itanium® instructions and other information needed to construct the code spaces and the initial data for a TNS/E native process.

TNS/E native process. A process initiated by executing a TNS/E native object file. Contrast with TNS process.

TNS/E native signal. A signal model available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. TNS/E native signals are used for error exception handling.

TNS/E native user library. A user library available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. A TNS/E native user library is implemented as a TNS/E native dynamic-link library (DLL).

TNSVU. A tool used to browse through TNS object files that have been accelerated by the TNS Object Code Accelerator (OCA). TNSVU displays Intel® Itanium® code in addition to TNS code.

Total Information Manager (TIM). See HP NonStop™ Technical Library (NTL).

TNS/R. See NonStop™ Series TNS/R.

TNS/R library. A TNS/R native-mode library. For a PIC-compiled application, TNS/R libraries can be dynamic-link libraries (DLLs) or hybridized native shared runtime libraries (SRLs). For an application that is not PIC compiled, TNS/R libraries can only be native SRLs.

TNS/R native mode. The operational environment in which native-compiled RISC instructions execute.

TNS/R native object code. The RISC instructions that result from processing program source code with a TNS/R native compiler. TNS/R native object code executes only on TNS/R systems, not on TNS systems.

TNS/R native object file. A file created by a TNS/R native compiler that contains RISC instructions and other information needed to construct the code spaces and the initial data for a TNS/R native process.

TNS/R native process. A process initiated by executing a TNS/R native object file.

TNS/R native shared run-time library (TNS/R native SRL). A shared run-time library (SRL) available to TNS/R native processes in the Guardian and OSS environments. TNS/R native SRLs can be either public or private. A TNS/R native process can have multiple public SRLs but only one private SRL.

TNS/R native signal. A signal model available to TNS/R native processes in the Guardian and OSS environments. TNS/R native signals are used for error exception handling.

TNS/R native user library. A user library available to TNS/R native processes in the Guardian and OSS environments. A TNS/R native user library is implemented as a special private TNS/R native shared run-time library.

transfer mode. The protocol by which data is transferred between a terminal and the computer system. See conversational mode and page mode.

trap. A software mechanism that stops program execution and holds the cause of a processing problem. In TNS and accelerated execution modes, traps occur as the result of errors that are not related to input or output. Contrast with signal.

trap handler. A location in a program where execution begins if a trap occurs. A process can specify a trap handler by a call to the Guardian ARMTRAP procedure.

two-way communication. A form of interprocess communication in which the sender of a message (requester process) expects data in the reply from the receiver (server process). Contrast with one-way communication.

UID. A nonnegative integer that uniquely identifies a user within a node.

In the Open System Services (OSS) environment, the UID is the scalar view of the HP NonStop™ operating system user ID. The UID is used in the OSS environment for functions normally associated with a UNIX user ID.

unlabeled tape. A magnetic tape that does not contain standard ANSI or IBM tape labels. Contrast with labeled tape.

unnamed process. A process to which a process name was not assigned when the process was created. Contrast with named process.

unstructured file. A disk file that has no structure known to the disk process. Contrast with structured file.

user code. A logically distinct part of the HP NonStop™ operating system that consists of the code for user processes.

user database. A database within an HP NonStop™ node that contains the user name, user ID, group ID, initial working directory, and initial user program for each user of the node.

user data stack. The first 64K bytes in the user data segment in a TNS process. It is the only 16-bit byte-addressable area. It is the primary area for static and dynamic data as well as for saving procedure call and return information.

user ID. The unique identification of a user within a node.

In the Guardian environment, the term *user ID* usually means the structured view of the HP NonStop™ operating system user ID. In the Open System Services (OSS) environment, the term *user ID* usually means the scalar view of the HP NonStop™ operating system user ID—a number called the UID.

User Library (UL). See TNS code space.

user library. (1) An object code file that the operating system links to a program file at run time. A program can have only one user library. See also TNS user library and TNS/E native user library.

user name. A string that uniquely identifies a user within the user database for a node.

user process. A process whose primary purpose is to solve a user's problem. A user process is not essential to the availability of a CPU. A user process is created only when the user explicitly creates it. Contrast with system process.

vertical form control (VFC) table. A table in the memory of a 5515, 5516, or 5518 printer that is used to control vertical forms movement.

VFC table. See vertical form control (VFC) table.

virtual memory. A range of addresses that processes use to reference real storage, where real storage consists of physical memory and disk storage.

volume. A disk drive or a pair of disk drives that forms a mirrored disk.

voluntary rendezvous opportunity (VRO). A point in the instruction stream of a processing element at which the operating system is permitted to establish synchronization among the CPU elements of a logical process. Most system calls are VROs.

waited I/O. An operation with an I/O device or process where the issuing process waits until the operation finishes. Contrast with nowait I/O.

waiting process. A process that cannot execute until an event occurs, a resource becomes available, or an interval of time passes.

wall-clock time. The local time-of-day, including any adjustment for daylight-savings time.

word. An instruction-set-defined unit of memory that corresponds to the width of registers and to the most common and efficient size of memory operations. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. An Intel® Itanium® word is also 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory.

working set. A collection of DEFINE attributes that have been assigned values using the DEFINESETATTR procedure. The working set becomes a DEFINE when the DEFINEADD procedure is called.

$CMON. Seecommand-interpreter monitor ($CMON).

$RECEIVE. A special file name through which a process receives messages from other processes.

Index

Numbers

32-bit ELF format 1-22

48-bit timestamps
  converting to Gregorian date and time 18-14
  definition 18-3
  obtaining 18-14
  working with 18-13/18-14

5515 matrix line printer, communicating with 11-2/11-9, 11-19/11-49

5516 matrix line printer, communicating with 11-2/11-9, 11-19/11-49

5518 matrix line printer, communicating with 11-2/11-9, 11-19/11-49

5573 Laser-LX printer, communicating with 11-2/11-19, 11-29/11-49

5573D Laser-LX printer, communicating with 11-2/11-19, 11-29/11-49

5574 laser printer, communicating with 11-2/11-19, 11-29/11-49

5577 laser printer, communicating with 11-2/11-19, 11-29/11-49

64-bit ELF format 1-22

64-bit Julian timestamps
  See Julian timestamps

A

abort C function 25-24

Accelerator 1-23, 16-4

Access mode, files
  See Files, access mode

Access mode, SIO 15-15/15-16, 15-67

Active backup programming
  buffered I/O 27-17
  C support of 27-5
  file synchronization 27-15
  introduction to 27-1
  overview 27-1/27-3
  program examples 27-32, 27-52

Active backup programming (continued)
  updating state information 27-11, 27-23

ADDDSTTRANSITION command 18-31/18-36

ADDDSTTRANSITION procedure 18-30/18-31

Adding users 23-24/23-26

ADDRESS_DELIMIT_ procedure 17-1, 17-15/17-16, 17-38

Adduser^msg message 23-24/23-26

Adduser^reply structure 23-25/23-26

alarm C function 25-24

ALLOCATE^FCB DEFINE 15-7/15-10

ALLOCATE^FCB^D00 DEFINE 15-7/15-10

Alternate-key files
  adding keys to 5-85
  creating 5-81/5-85
  entry-sequenced primary file 5-72
  key-sequenced primary file, example 5-72/5-76
  purpose of 5-7
  relative primary file, example 5-71, 5-86/5-88
  structure of 5-7/5-8

Altpri^msg message 23-22/23-24

Altpri^reply structure 23-23/23-24

Ancestor process 16-11/16-12

Application design, fault-tolerant 1-2

Application design, requester/server model
  See Requester/server model

Approximate positioning 5-49

ARMTRAP procedure 1-10, 25-11/25-12, 25-13/25-15, 25-29

ASSIGN 1-7

ASSIGN command
  described 8-2
  with SIO 15-15/15-19

Assign message
  example 8-10
  introduction to 8-8/8-10
  processing 8-12/8-13
  receiving 8-12, 16-46
  sending 16-46
  structure of 8-9
ASSIGN^BLOCKLENGTH operation 15-18/15-19
ASSIGN^FILECODE operation 15-17
ASSIGN^FILENAME operation 15-67
ASSIGN^LOGICALFILENAME operation 15-10
ASSIGN^OPENACCESS operation 15-15/15-16, 15-38
ASSIGN^OPENEXCLUSION operation 15-16, 15-16
ASSIGN^PRIMARYEXTENTSIZE operation 15-18
ASSIGN^RECORDLENGTH operation 15-17
ASSIGN^SECONDARYEXTENTSIZE operation 15-18
Asterisk (*) 13-3/13-4
Attributes, of DEFINEs
  See DEFINE attributes
AWAITIO procedure 4-4/4-23
  EDIT files 14-18
  IOEdit 14-18
  magnetic tape 12-2
  printers 11-2
  SIO files 15-42/15-45
  terminals 10-2
  $RECEIVE 6-2
AWAITIOX procedure 4-4/4-23
  extended data segments 17-35/17-36
  magnetic tape 12-2
  printers 11-2
  terminals 10-2, 10-8
  timelimit parameter 10-8
  $RECEIVE 6-2

B

Backslash (\), in node name 2-5
Backspace character 10-13
BACKSPACEEDIT procedure 14-17
Batch processing
  assigning job-ID 16-35/16-36
  relationship between processes 16-13/16-14
Binary semaphores 1-24
  closing 26-8
  creating 26-5
  example 26-10
  forcing a lock 26-8
  Guardian procedure calls 26-4/26-8
  locking 26-6
  opening 26-6
  overview 26-1/26-3
  unlocking 26-7
Binder program 25-7
BINSEM_CLOSE_ procedure 26-8
BINSEM_CREATE_ procedure 26-5
BINSEM_FORCELOCK_ procedure 26-8
BINSEM_LOCK_ procedure 26-6
BINSEM_OPEN_ procedure 26-6
BINSEM_UNLOCK_ procedure 26-7
Blank characters
  ASCII
    padding before writing 15-33/15-34
    trimming before writing 15-32/15-33
    trimming on reading 15-34/15-35
  MBCS 19-54
Block buffering, of SIO files 15-25/15-26
Block length, SIO 15-67/15-71
Block mode
  See Page mode
Blocking records 12-15/12-16, 12-26/12-27, 12-72
BLOCKLEN attribute 12-26
BREAK key 10-23/10-25

BREAK mode 10-27/10-36
BREAK ownership
  releasing 10-26, 15-56
  taking 10-25/10-26, 15-55/15-56
BREAKMESSAGE_SEND_ procedure 24-15
Breakpoints 25-6
Break-on-device message 24-15
Buffered mode
  end-of-file marks 12-19
  errors, recovering from 12-85/12-86
  example of 12-19/12-20
  introduction to 12-16/12-17
  invoking 12-17/12-18
  labeled tape 12-29
  revoking 12-18
  streaming devices 12-19
  unlabeled tape 12-74

C

C and C++, calling Guardian procedures from 1-19
Cache flushes, minimizing 5-13
CAID 16-10/16-11
CANCEL procedure 6-2, 6-23
  printers 11-2
  terminals 10-2
Canceled messages 6-23/6-25
Cancellation messages 6-24/6-25
CANCELPROCESSTIMEOUT procedure 18-17/18-18
CANCELREQ procedure 6-2, 6-23
  printers 11-2
  terminals 10-2
CANCELTIMEOUT procedure 18-16/18-17
Case shifting
  ASCII characters 19-36/19-37
  multibyte characters 19-55
CASECHANGE procedure 19-55
CCE 2-27
CCG 2-27
CCL 2-27
cextdecs header 1-19
Character sets 1-9, 19-49/19-65
Checkpointing 1-2, 1-8, 1-15
CHECK^BREAK procedure 15-2, 15-56
CHECK^FILE procedure 15-2, 15-3, 15-26/15-27
Child process, monitoring 16-46/16-47
CHILD_LOST_ procedure 16-1, 16-46/16-47
Chinese Big 5 character set 19-49
Chinese PC character set 19-49
CI messages
  See Command interpreter messages
CISC Glossary-3
CLASS attribute
  labeled tape 12-23
  of DEFINEs 7-9
  unlabeled tape 12-70
CLASS DEFAULTS DEFINEs
  See DEFAULTS CLASS DEFINEs
CLASS MAP DEFINEs
  See MAP CLASS DEFINEs
CLASS SEARCH DEFINEs
  See SEARCH CLASS DEFINEs
CLASS TAPE DEFINEs
  See TAPE CLASS DEFINEs
CLEAR command 8-2
Client process
  See Requester process
Clock averaging 18-1/18-2
Close message 6-26, 6-29, 15-53, 22-7
CLOSEALLEDIT procedure 14-1, 14-19
CLOSEEDIT procedure 14-1, 14-19
CLOSE^FILE procedure 15-2, 15-65
COBOL85, calling Guardian procedures from 1-20
COBOLEX0 file 1-20
COBOLEX1 file 1-20
COBOLEXT file 1-20

Code spaces 16-2, 16-5/16-7
Command interpreter
  See TACL
Command interpreter messages
  number -1 (Startup)
    example 8-5
    introduction to 8-3/8-4
    processing 8-6/8-8
    reading 8-6
    See also Startup message
    structure of 8-4
    terminal files 10-4/10-5
  number -2 (Assign)
    example 8-10
    introduction to 8-8/8-10
    processing 8-12/8-13
    reading 8-12
    See also Assign message
    structure of 8-9
  number -20 (Wakeup) 8-19/8-20
  number -21 (Display) 8-20/8-21
  number -3 (Param)
    example 8-11
    introduction to 8-8/8-11
    processing 8-14/8-15
    reading 8-12
    See also Param message
    structure 8-11
  number -50 (Logon^msg) 23-10/23-11
  number -51 (Logoff^msg) 23-12
  number -52 (Processcreation^msg) 23-17/23-22
  number -53 (Illegal^logon^msg) 23-13/23-14
  number -54 (Adduser^msg) 23-24/23-26
  number -55 (Deluser^msg) 23-26/23-27
  number -56 (Altpri^msg) 23-22/23-24
  number -57 (Password^ msg) 23-14/23-15
  number -58 (Remotepassword^msg) 23-15/23-17
  number -59 (Prelogon^msg) 23-9/23-10
  number -60 (Config^msg) 23-4/23-8
Command interpreter monitor
  See $CMON
Common FCB 15-6, 15-66
Common file control block 15-6, 15-66
COMPLETEIOEDIT procedure 14-1, 14-18
Completion codes 16-49/16-50
Complex instruction-set computing (CISC) Glossary-5
COMPRESSEDIT procedure 14-1, 14-18
COMPUTEJULIANDAYNO procedure 18-13
COMPUTETIMESTAMP procedure 18-10/18-11
Concurrency control
  See Files, concurrent access
Concurrent message processing, example 6-22
Condition codes 2-27/2-28
  magnetic tape
    from CONTROL operations 12-3
    from SETMODE functions 12-3
  printers
    from CONTROL operations 11-3
    from SETMODE functions 11-3
  terminals
    from CONTROL operations 10-3
    from SETMODE functions 10-4
Configuring TACL 23-4/23-8
Config^msg message 23-4/23-8
Config^reply structure 23-7/23-8
Config^text^reply structure 23-7
Context-free server 22-3
CONTIME procedure 18-14
Control message 6-26, 6-30, 24-10

CONTROL procedure 2-23
    device operations 9-4
    interprocess communication 6-2
    magnetic tape operations 12-2/12-3
    operation-1
        printers 11-3, 11-21
        terminals 10-3, 10-17/10-18
    operation-10 12-3, 12-8/12-10
    operation-11 10-3, 11-3
    operation-12 10-3, 11-3
    operation-2 12-2, 12-13
    operation-24 12-3
    operation-26 12-18
    operation-27 5-33/5-35
    operation-3 12-2, 12-10
    operation-4 12-2, 12-10
    operation-5 12-2, 12-11
    operation-6 12-2, 12-11
    operation-7 12-2, 12-5
    operation-8 12-2, 12-5/12-10
    operation-9 12-2, 12-7/12-8
    printer operations 11-3
    terminal operations 10-2, 10-3
Controlbuf message 6-26, 6-30
CONTROLBUF procedure 6-2
CONTROLBUF procedure, device operations 9-4
Controlling openers 6-29
Conversational mode
    form feed 10-17/10-18
    interrupt characters
        See Interrupt characters
    line spacing 10-16/10-17
    line-termination character 10-11/10-12
    overview 10-10/10-11
    setting 10-9/10-10
CONVERTPROCESSTIME procedure 18-19
CONVERTTIMESTAMP procedure 18-11
CPU down message 15-53, 22-7/22-8
CPU failure 22-7/22-8
Creator access ID 16-10/16-11
Current-record pointer 2-18/2-23

D

Data descriptors, formatter 19-7
Data integrity 2-16
Data segment, extended 1-9
Data segment, user 1-8
Data spaces 16-2, 16-8/16-10
Data stack 17-3/17-6
Data, manipulating 1-9
Daylight-saving time 18-2
DAYOFWEEK procedure 18-13
DCT 16-2, 16-11/16-12
Deadlocks
    avoiding 3-8, 3-10
    examples 3-7/3-11
DEBUG command 25-5/25-6
DEBUG procedure 25-3
Debug program 1-9, 25-1/25-9
Debug state 16-18/16-19, 25-2/25-6
Debugger
    Debug program 1-9, 25-1/25-9
    high level 1-9, 25-1/25-9
    Inspect program 1-9, 25-1/25-9
    low level 1-9, 25-1/25-9
Debugging
    switching debuggers during a debugging session 25-7
Debugging tools 25-1
    Native Inspect 25-2
    Visual Inspect 25-1
Debugging, environment for 25-6/25-9
Decorations, formatter 19-6/19-7
DEFAULTS CLASS DEFINEs
    modified by file system 7-6
    purpose 7-3, 7-5/7-6

DEFINE attributes
  errors 7-13
  restoring 7-13
  setting 7-12/7-13
DEFINE errors
  number-2057 7-14
  number-2058 7-14
  number-2059 7-14
DEFINE mode 7-10
DEFINE names, resolving 13-9/13-11
DEFINEADD procedure 7-1, 7-14/7-15, 12-22
DEFINEDELETE procedure 7-1, 7-15
DEFINEDELETEALL procedure 7-1, 7-15
DEFINEMODE procedure 7-11, 12-22, 12-70
DEFINERESTORE procedure 7-1, 7-13, 7-15
DEFINERESTOREWORK procedure 7-1, 7-16/7-17
DEFINERESTOREWORK2 procedure 7-1, 7-16/7-17
DEFINEs
  adding 7-11/7-17
  attributes 7-8/7-10
    data types 7-8/7-9
    values 7-9
  CLASS attribute 7-9
  DEFAULTS CLASS
    See DEFAULTS CLASS DEFINEs
  definition of 1-6
  deleting 7-15
  enabling 7-10/7-11
  example 7-17/7-40
  introduction to 7-1/7-2
  labeled tape 12-22/12-28
  MAP CLASS
    See MAP CLASS DEFINEs
  naming 7-7/7-8
  propagating 7-10/7-11, 16-36/16-37
  purpose of 7-2/7-6
DEFINEs (continued)
  referring to 7-11
  restoring 7-15/7-16
  saving 7-15/7-16
  SEARCH CLASS
    See SEARCH CLASS DEFINEs
  See also Working set
  TAPE CLASS
    See TAPE CLASS DEFINEs
  unlabeled tape 12-70/12-71
  =_DEFAULTS 2-5, 7-3, 7-5/7-6
DEFINESAVE procedure 7-1, 7-15
DEFINESAVEWORK procedure 7-1, 7-16/7-17
DEFINESAVEWORK2 procedure 7-1, 7-16/7-17
DEFINESETATTR procedure 7-1, 7-12/7-13
  BLOCKLEN attribute 12-26
  CLASS attribute 12-23, 12-70
  DENSITY attribute 12-27, 12-72
  DEVICE attribute 12-71
  EBCDIC attribute 12-25
  FILEID attribute 12-24
  FILESECT attribute 12-24
  FILESEQ attribute 12-24
  LABELS attribute 12-23, 12-70
  RECFORM attribute 12-27
  RECLEN attribute 12-26
  REELS attribute 12-24
  SYSTEM attribute 12-24
  USE attribute 12-25
  VOLUME attribute 12-24
DEFINESETLIKE procedure 7-1, 7-13
DEFINESETMODE procedure 7-1
DEFINEVALIDATEWORK procedure 7-1, 7-13
DELETEEDIT procedure 14-1, 14-16/14-17
Deleting users 23-26/23-27
Deluser^msg message 23-26/23-27
Deluser^reply structure 23-27

DENSITY attribute
  labeled tape 12-27
  unlabeled tape 12-72
Destination control table (DCT) 16-2, 16-11/16-12
DEVICE attribute
  labeled tape 12-27
  unlabeled tape 12-71
Device files
  home terminal
    getting the name of 9-3
    opening 2-17
  I/O with 2-23
  naming 2-6, 9-2/9-3
  opening 2-16
  See also Magnetic tape
  See also Printers
  See also Terminals
Device mode
  labeled tape 12-28
  unlabeled tape 12-73
Device names
  searching for 13-23/13-24
  syntax of 2-6/2-7
Device subtypes 16-35
Device subtypes, subtype 30 24-2
Devices
  communicating with 9-1/9-11
  getting information about 9-5/9-11
  I/O with 1-6
  magnetic tape, I/O with 1-7
  printers, I/O with 1-7
  See also Device files
  terminals, I/O with 1-6
DEVICE_GETINFOBYLDEV_ procedure 9-5/9-11
  printers 11-2
DEVICE_GETINFOBYNAME_ procedure 9-5/9-11
  printers 11-2
Disk files
  accessing 1-5
  alternate-key
    See Alternate-key files
  creating 2-11/2-12
  entry-sequenced
    See Entry-sequenced files
  extents, allocating 2-12
  introduction to 1-5
  I/O with 2-18/2-23
    example 2-29/2-38
  key-sequenced
    See Key-sequenced files
  naming 2-4, 2-6
  opening 2-16
  partitioned
    See Partitioned files
  permanent
    See Permanent disk files
  queue
    See Queue files
  random access 2-21
  relative
    See Relative files
  structured 2-2
  subvolumes 2-2
  syntax of 2-4/2-6
  temporary
    See Temporary disk files
  TM/MP 1-5
  unstructured
    See Unstructured files
  volumes 2-2
Disk I/O process 2-16
Display message 8-20/8-21
DLLs
  implicit 16-4, 16-5
DNUMIN procedure 19-1, 19-35/19-36
DNUMOUT procedure 19-1, 19-36

Dollar sign ($)
  in device name 2-7
  in process name 2-7
  in volume name 2-5
DST transition table 18-29/18-36
Duplicate keys 5-8

E

E register 17-4
EBCDIC attribute 12-25
Edit descriptor 19-6/19-7
EDIT file segment (EFS) 14-7
EDIT files
  accessing with SIO 15-37/15-39
  appending to with IOEdit 14-14
  closing with IOEdit 14-19
  compressing with IOEdit 14-18
  creating for IOEdit 14-9/14-10
  deleting lines with IOEdit 14-16/14-17
  file pointer positioning 14-12/14-13
  initializing for IOEdit 14-10
  inserting lines with IOEdit 14-15
  IOEdit access 14-1/14-19
  line backspacing with IOEdit 14-17
  nowait I/O and IOEdit 14-18
  opening for IOEdit 14-8/14-10
  opening for SIO 15-38
  packed line format 14-5/14-7
  reading from SIO 15-38
  reading with IOEdit 14-11/14-17
  renumbering lines with IOEdit 14-15/14-16
  restrictions 14-1/14-2
  sequential reading with IOEdit 14-13
  sequential writing with IOEdit 14-13/14-15
  setting the read position for SIO 15-38/15-39
  writing with IOEdit 14-11/14-17
  writing with SIO 15-38
EFS 14-7
Elapsed time 18-15
  canceling timers 18-16/18-17
  measuring long intervals 18-20/18-24
  setting timers 18-16/18-17
ELF format 1-22
End-of-file character 10-14
End-of-file pointer 2-18
Enscribe files
  See Disk files
Entry-sequenced files
  altering attributes of 5-33
  alternate key access 5-72
  closing 5-33
  creating 5-31/5-32
  example of programming with 5-36/5-46
  introduction to 2-2
  I/O with 5-32/5-33
  locking 5-33
  opening 5-32
  positioning 5-32/5-33
  purging 5-33
  purpose of 5-4
  reading 5-32/5-33
  renaming 5-33
  structure of 5-4/5-5
  writing 5-32/5-33
Environment register 17-4
Error number, file system 2-26/2-28
ERROR process
  closing 20-5
  creating 20-2/20-3
  deleting 20-5
  example program 20-6/20-12
  opening 20-3
  overview of 20-1
  reading messages from 20-5
  running 20-2/20-3
  sending a Startup message to 20-4

Errors
  DEFINE
    See DEFINE errors
  file system
    See File system errors
  nonretryable 2-28
  process creation 16-27/16-28
  retryable 2-28
  SIO
    See SIO errors
Exact positioning 5-49
Exclusion mode, files
  See Files, exclusion mode
Exclusion mode, SIO 15-16, 15-67/15-71
Execution modes, TNS/R 1-23
EXPIRATION attribute 12-27
EXTDECS file 1-19
EXTDECS0 file 1-18, 1-19
EXTDECS1 file 1-19
Extend access 3-1
Extended data segments 1-9
  allocating 17-23
  bounds checking 17-34, 17-38
  deallocating 17-42
  extensible 17-23, 17-27
  getting the size of 17-34
  introduction to 17-16/17-22
  I/O with 17-35/17-37
  large buffer transfers 17-36/17-37
  moving data between 17-37
  read only 17-23, 17-27
  referencing data in 17-29/17-33
  resizing 17-34
  shared 17-23, 17-27, 17-39/17-41
Extended memory
  See Extended data segments
EXTENDEDIT procedure 14-1, 14-16
Extents
  allocating 2-12
  primary 2-12
  secondary 2-12
  SIO 15-18, 15-67/15-71
  sizes 2-12
External declarations file
  C and C++ 1-19
  COBOL85 1-20
  Pascal 1-20
  TAL and pTAL 1-18

F

Fault tolerance 1-1
  active backup 27-1
  data integrity 2-16
  fault containment 1-3
  monitor 1-15
  multiple copies of operating system 1-3
  Pathway 1-8
  persistent process 1-2, 1-15
  process pairs 1-2, 1-8, 27-1/27-2
  requester/server model 1-15
  sync depth 2-16
  system integrity 1-3
  TM/MP 1-2, 1-8
fault tolerance, mirrored disk 1-3
Fault-tolerant programming
  See Active backup programming
FCBSIZE literal 15-66
FCBSIZE^D00 literal 15-66
FCBs, SIO 15-2
  allocating 15-7/15-10, 15-66
  initializing 15-7/15-10, 15-67
  naming 15-7/15-10, 15-67
File access mode, SIO 15-15/15-16, 15-67/15-71
File code, SIO 15-17, 15-67/15-71
File control blocks (FCBs), SIO
  See FCBs, SIO

File control blocks, SIO
  See FCBs, SIO
File names 1-4
  comparing 13-20/13-21
  default values 2-5, 13-8
  device 2-6
  disk
    location-independent 2-9
    permanent 2-4/2-6
    temporary 2-6
  editing 13-18/13-20
  extracting a part of 13-15/13-16
  extracting a prefix from 13-16
  extracting a suffix from 13-16
  extracting pieces of 13-15/13-18
  extracting pieces without implicit resolution 13-16/13-17
  fully qualified 2-5, 13-7/13-13
  manipulating 13-1/13-41
  matching
    complete match 13-31/13-32
    examples 13-33
    incomplete match 13-32/13-33
  modifying 13-18/13-20
  part
    definition of 13-2
  partially qualified 2-5
  part, examples of 13-2
  passing to processes 2-9
  piece
    definition of 13-3
    examples of 13-3
  process
    named 2-7
    unnamed 2-8
  process descriptors 2-8
  resolving 13-7/13-13
  scanning 13-4/13-6
File names (continued)
  searching for and resolving 13-11/13-12
  subpart 13-3
  truncating 13-13/13-15
  validating 13-4/13-6
  wild-card characters in 13-3/13-4
  * symbol 13-3/13-4
  ? symbol 13-3/13-4
File number 2-16
File pointers
  current record 2-18/2-23
  end-of-file 2-18
  next record 2-18/2-23
File system 1-4/1-7
  buffering 4-8/4-9
  error handling 2-26/2-28
  errors
    nonretryable 2-28
    number 30-to-39 10-37
    number 638 16-51/16-52
    number 639 16-51/16-52
    number-100 11-30, 12-87, 12-87/12-88, 15-63/15-64
    number-101 15-63/15-64
    number-102 11-30, 15-63/15-64
    number-103 15-64
    number-110 10-37/10-38, 15-64
    number-111 10-37/10-38, 15-64
    number-112 10-38, 15-64
    number-124 15-64
    number-140 10-38
    number-153 12-87/12-88
    number-154 12-6, 12-8
    number-200 6-31
    number-201 6-31
    number-21 12-13
    number-218 12-87/12-88
    number-40 6-31, 10-37
    number-45 14-16

File system (continued)
    path errors 10-38/10-39, 11-30, 12-52, 12-88, 15-65
    printers 11-29/11-31
    retryable 2-28
    terminals 10-36/10-41
File System I/O Synchronization 21-4
FILEID attribute 12-24
FILEINFO command 3-2/3-3
FILENAME_COMPARE_ procedure 13-1, 13-20/13-21
FILENAME_DECOMPOSE_ procedure
    file-name part 13-15/13-16
    file-name prefix 13-16
    file-name suffix 13-16
    introduction to 13-1, 13-15
    no-defaults flag 13-16/13-17
    process descriptor 13-17/13-18
FILENAME_EDIT_ procedure 13-1, 13-18/13-20
FILENAME_FINDFINISH_ procedure 13-1, 13-31
FILENAME_FINDNEXT_ procedure 13-1, 13-26/13-31
FILENAME_FINDSTART_ procedure 13-1, 13-21/13-26
FILENAME_MATCH_ procedure 13-1, 13-31/13-33
FILENAME_RESOLVE_ procedure
    defaults override 13-8
    DEFINE names 13-9/13-11
    file name override 13-12/13-13
    file names 13-7/13-8, 13-11/13-12
    introduction to 13-1
    subvolume names 13-8/13-9
FILENAME_SCAN_ procedure
    examples 13-6
    file names 13-5
    file-name patterns 13-6
    introduction to 13-1, 13-4/13-5
    node names 13-5
FILENAME_SCAN_ procedure (continued)
    subvolume names 13-6
FILENAME_TO_PROCESSHANDLE_ procedure 16-1, 16-62/16-63
FILENAME_UNRESOLVE_ procedure
    all default values 13-13/13-14
    alternate default values 13-14/13-15
    introduction to 13-1, 13-13
    subset of default values 13-14
Files
    access mode 3-1/3-3
        example 3-2
    closing 2-29
    concurrent access 1-5, 3-1/3-11
    creating 2-10/2-15
    deadlocks 3-7/3-11
    Enscribe 1-5
    exclusion mode 3-3/3-6
        exclusive 3-4
        protected 3-4
        shared 3-3
    getting information on 2-26
    I/O with 1-4, 2-18/2-25, 4-1/4-23
    locking 1-5, 3-6/3-11
    naming 2-4/2-9
    opening 2-15/2-18
        for nowait I/O 4-3
        nowait I/O 4-5
    ownership 3-2/3-3
    security 3-2/3-3
    See also Device files
    See also Device names
    See also Disk files
    See also File names
    See also Process files
    See also $RECEIVE
FILESECT attribute 12-24
FILESEQ attribute 12-24

File-name patterns
  introduction to 13-3/13-4
  matching 13-21/13-33
  scanning 13-6
  searching for 13-21/13-33
  validating 13-6
FILE_ALTERLIST_ procedure 5-15
FILE_CLOSE_ procedure
  all files 2-29
  magnetic tape 12-2, 12-84
  printers 11-2, 11-4/11-5
  processes 6-2
  terminals 10-2, 10-10
  unstructured files 5-13
  $RECEIVE 6-13
FILE_CREATELIST_ procedure
  alternate-key file 5-82/5-85
  disk files 2-11
  entry-sequenced files 5-31/5-32
  extent sizes, specifying 2-13
  key-sequenced files 5-47/5-48
  partitioned file 5-78/5-80
  relative files 5-16/5-17
  unstructured files 5-9/5-10
FILE_CREATE_ procedure 5-48
  disk files 2-11/2-13
  entry-sequenced files 5-31/5-32
  key-sequenced files 5-47
  relative files 5-16/5-17
  unstructured files 5-9/5-10
FILE_GETINFOBYNAME_procedure 2-26, 10-2
FILE_GETINFOLISTBYNAME_ procedure 2-26
FILE_GETINFOLIST_ procedure 2-26
FILE_GETINFO_ procedure
  all files 2-26, 2-27
  magnetic tape 12-2
  printers 11-2
  process files 6-3
  terminals 10-2
  $RECEIVE 6-3
FILE_GETRECEIVEINFO_ procedure 6-3, 6-18/6-21
FILE_OPEN_ procedure
  all files 2-15/2-18
  EDIT files 14-8
  entry-sequenced files 5-32
  home terminal 10-6
  IN file 10-4/10-5
  key-sequenced files 5-48
  labeled tape 12-22, 12-28
  magnetic tape 12-2, 12-4
  nowait parameter 4-3, 4-5
  OUT file 10-4/10-5
  partitioned files 5-80
  printers 11-2, 11-4/11-5
  process files 6-3
  processes 6-4/6-5
  receive-depth parameter 6-10, 6-17
  relative files 5-17
  sync-depth parameter of 2-16
  terminal simulator 24-3
  terminals 10-2, 10-4/10-6
  unlabeled tape 12-71
  unstructured files 5-11
  $RECEIVE 6-3, 6-15
  $RECEIVE for reading system messages 6-27
FILE_PURGE_ procedure 5-14
FILE_RENAME_ procedure 5-12
FIXSTRING procedure
  example of 19-39/19-46
  overview of 19-1, 19-37
  template for 19-38/19-39

Flat segments
  allocating 17-24
  introduction to 17-19/17-23
  referencing data in 17-32/17-33
Form feed
  printers 11-21/11-25
  terminals 10-17/10-18
FORMATCONVERT procedure 19-1/19-34
FORMATCONVERTX procedure 19-1/19-34
FORMATDATA procedure 19-1/19-34
FORMATDATAX procedure 19-1/19-34
Formatter 19-1/19-34
  using in TNS/R native programs 19-2
Formatting
  buffer control 19-11/19-13
  data descriptors 19-7
  edit descriptors 19-6/19-7
  format-directed input 19-4/19-6
  format-directed output 19-3/19-4
    example 19-19/19-25
  list-directed input 19-26/19-34
    example 19-27/19-34
  list-directed output 19-34
  literals 19-13/19-15
  numbers 19-8/19-10
  plus control 19-18/19-19
  scale factor 19-18
  tabulated data 19-15/19-18
  text 19-8/19-10
FORTRAN, calling Guardian procedures from 1-20
FUP ALTER command 2-13, 5-15
FUP CREATE command
  alternate-key files 5-81/5-82
  key-sequenced files 5-47/5-48
  partitioned files 5-78
  unstructured files 5-9/5-10
FUP PURGE command 5-14

G

GEN attribute 12-28
Generic key positioning 5-50
GETINCREMENTEDIT procedure 14-1, 14-15
GETPOSITIONEDIT procedure 14-1, 14-15, 14-16/14-17
GIVE^BREAK procedure 15-2, 15-56
Globals-heap segment 1-8, 16-9, 16-10, 17-6, 17-8
GMT
  converting to local time 18-11
  definition of 18-2
GPLDEFS file 15-3, 15-4, 15-5, 15-6, 15-65
Greenwich mean time (GMT)
  See GMT
Gregorian date and time
  converting to Julian timestamp 18-10/18-11
  obtaining from Julian timestamp 18-10
Gregorian dates
  converting to Julian day number 18-13
  obtaining from Julian day number 18-13

H

Hangul character set 19-49
Hash sign
  See Pound sign
HEADROOM_ENSURE_ procedure 17-1, 17-14
Heap 1-8, 16-9, 17-13
HEAPSORT procedure 19-1, 19-46
HEAPSORTX_ procedure 19-1, 19-46/19-49
Heap, sizing 17-13
High PIN 16-3, 16-28/16-31
HIST_FORMAT_ procedure 25-23, 25-32
HIST_GETPRIOR_ procedure 25-23, 25-32

HIST_INIT_ procedure 25-23, 25-32
Home terminal
  getting the name of 9-3
  opening 2-17, 10-6
  relationship of process with 16-14
  specifying 16-31/16-32
H-series
  debugging tools 25-1

I

IBM Kanji character set 19-50
IBM Kanji Mixed character set 19-50
Illegal logon, controlling from $CMON 23-13/23-14
Illegal^logon^msg message 23-13/23-14
Illegal^logon^reply structure 23-13/23-14
Implicit DLLs 16-4, 16-5
IN file 9-3
  and the SYSTEM command 8-2
  contents of 8-4
  opening 10-4/10-5
  with SIO 15-7/15-10
INCREMENTEDIT procedure 14-1, 14-14, 14-15, 14-15
Index blocks 5-5
INITIALIZEEDIT procedure 14-8
INITIALIZER procedure 8-3, 8-6/8-8, 8-12/8-16
  differences between native and TNS 15-4, 15-13
  SIO applications 15-5/15-22
INIT^FILEFCB operation 15-67
INIT^FILEFCB^D00 operation 15-67
Input/output
  See I/O
Inspect program 1-9, 25-1/25-9
Inspect state 16-18/16-19
INSPECT TAL compiler directive 25-7
INTERPRETINTERVAL procedure 18-11/18-12
INTERPRETJULIANDAYNO procedure 18-13
INTERPRETTIMESTAMP procedure 18-10, 18-12
Interprocess communication
  canceled messages 6-23/6-25
  closing $RECEIVE 6-13
  example of 6-31/6-63
  getting message information 6-19/6-21
  handling concurrent messages 6-17/6-22
  handling errors 6-31
  introduction to 1-6, 6-1/6-4
  message tag 6-21
  one way 6-15/6-16
  opening a process 6-4/6-5
  opening $RECEIVE 6-10, 6-15, 6-17
  queuing messages on $RECEIVE 6-8/6-9
  receiving messages 6-11, 6-15, 6-18, 15-46, 15-48/15-51
  reply count 6-20
  replying to messages 6-11/6-12, 6-21/6-22, 15-46
  sending messages 6-4/6-7
    no reply 6-6/6-7, 15-48
    reply expected 6-7, 15-46
  system messages 6-26/6-30
  two-way 6-10/6-14
Interrupt characters 10-20/10-22
  backspace 10-13
  changing 10-14/10-15
  conversational mode 10-12/10-16
  end-of-file 10-14
  line-cancel 10-13
  line-termination 10-11/10-12
  overview 10-12/10-13
  transparent mode 10-15
Interval timing 1-9, 18-7/18-8

IOEdit
- appending to files 14-14
- closing files 14-19
- compressing files 14-18
- creating files 14-9/14-10
- current-record pointer 14-11
- deleting lines 14-16/14-17
- EDIT file segment 14-7
- EFS 14-7
- file full errors 14-16
- file pointer positioning 14-12/14-13
- initializing files 14-10
- inserting lines 14-15
- introduction to 1-7, 14-1
- line backspacing 14-17
- line numbers 14-4/14-5
- next-record pointer 14-11
- nowait I/O with 14-18
- opening files 14-8/14-10
- packed line format 14-5/14-7
- reading 14-11/14-17
- record number increment 14-15
- record numbers 14-5, 14-11/14-12
- renumbering lines 14-15/14-16
- restrictions 14-1/14-2
- sequential reading 14-13
- sequential writing 14-13/14-15
- SIO, comparison with 14-2/14-4
- using 14-1/14-19
- writing 14-11/14-17

IOP 9-1

IPC
- one-and-a-half-way communication 2-25
- one-way communication 2-25
- See also Interprocess communication
- two-way communication 2-24

I/O
- nowait 1-5
  - See Nowait I/O
- to extended data segments 17-35/17-37
- waited 1-5

I/O process (IOP) 9-1

I/O subsystem 9-1/9-2

I'm Alive message 1-3

J

Jacket procedure 1-19

JEF (Fujitsu) Kanji character set 19-50

JEF (Fujitsu) Kanji Mixed character set 19-50

JIS Kanji character set 19-50

Job ancestor 16-13/16-14

Julian day numbers
- converting to day of the week 18-13
- converting to Gregorian dates 18-13
- introduction to 18-12
- obtaining from Gregorian dates 18-13
- obtaining from Julian timestamp 18-12

Julian timestamps
- converting to Gregorian date and time 18-10
- definition 18-2/18-3
- obtaining
  - local node 18-6/18-7
  - remote node 18-8/18-10
- obtaining from Gregorian date and time 18-10/18-11
- timing an interval with 18-7/18-8

JULIANTIMESTAMP procedure 18-6/18-14

K

Kanji character set 19-49
Kernel Managed Swap Facility Glossary-15
Kernel Managed Swap Facility (KMSF)
  requesting space guarantee from 16-34/16-35
Key compaction 5-47
Key compression 5-47
Key length 5-47
Key offset 5-47
Key specifier 5-7
KEYPOSITION procedure
  approximate positioning 5-49
  exact positioning 5-49
  generic key positioning 5-50
Key-sequenced files
  See also Queue files
Key-sequenced files
  altering attributes of 5-51
  alternate key access 5-72/5-76
    example 5-74/5-76
  approximate positioning 5-49
  closing 5-51
  creating 5-47/5-48, 5-81/5-85
  exact positioning 5-49
  example of programming with 5-51/5-67
  generic key positioning 5-50
  introduction to 2-2
  I/O with 5-49/5-51
  key compaction 5-47
  key compression 5-47
  key length 5-47
  key offset 5-47
  locking 5-51
  opening 5-48
  purging 5-51
  purpose of 5-5
  renaming 5-51
Key-sequenced files (continued)
  See also Alternate-key files
  structure of 5-5/5-7
kill C function 25-24
KMSF
  See Kernel Managed Swap Facility
KSC5601 character set 19-49

L

L register 17-4
Labeled tape
  accessing 12-22/12-29
  backward spacing by files 12-5/12-10
  backward spacing by records 12-8/12-10
  block size, specifying 12-26/12-27
  buffered mode 12-29
  closing 12-84
  conversion mode, selecting 12-25
  creating files 12-21
  device mode, selecting 12-28
  enabling processing of 12-21
  errors, recovering from 12-52, 12-85/12-89
  example program 12-52/12-89
  forward spacing by files 12-5
  forward spacing by records 12-7/12-8
  introduction to 12-1, 12-21
  I/O operation, specifying 12-25
  label type, specifying 12-23
  positioning 12-4/12-11
  reading from multiple-file tape 12-45/12-48
  reading from multiple-volume file 12-48/12-69
  reading from only file 12-42/12-45
  reading, concepts of 12-11/12-13
  record size, specifying 12-26/12-27
  rewinding 12-10/12-11
  support for, checking 12-21

Labeled tape (continued)
  tape density, specifying 12-27
  tape file, specifying 12-24
  tape volume, specifying 12-24
  writing to multiple-file tape 12-33/12-38
  writing to multiple-volume file 12-38/12-42
  writing to only file 12-29/12-33
  writing to, concepts of 12-13/12-14
labeled tape
  opening 12-4
LABELEDTAPESUPPORT procedure 12-21
LABELS attribute
  labeled tape 12-23
  unlabeled tape 12-70
Laser printers, communicating with 11-2/11-19, 11-29/11-49
LCT
  definition 18-2
  obtaining from GMT timestamp 18-11
Libraries, supplied by HP 16-4, 16-5
Library file 16-33
  as a private SRL 16-33
Line-cancel character 10-13
Line-termination character 10-11/10-12
Load balancing 1-14
Local civil time (LCT)
  See LCT
Local data address register 17-4
Local standard time (LST)
  See LST
LOCKFILE procedure 3-6/3-11
Locking
  files 3-6/3-11
  introduction to 1-5
  records 5-11
LOCKREC procedure 5-11
Logical device number 2-7
Logical file name, assigning 15-10
Logical-record length, SIO 15-17
Logoff, controlling from $CMON 23-12
Logoff^msg message 23-12
Logoff^reply structure 23-12
Logon, controlling from $CMON 23-9/23-11
Logon^msg message 23-10/23-11
Logon^reply structure 23-11
Long time intervals, measuring 18-20/18-24
LONGJMP_ procedure/C function 25-24, 25-26
Loss-of-communication-with-a-network-node message 15-53
Low PIN 16-3, 16-28/16-31
LST
  definition 18-2
  obtaining from GMT timestamp 18-11

M

Magnetic tape
  accessing 12-2/12-4
  backward spacing by files 12-5/12-10
  backward spacing by records 12-8/12-10
  blocking records 12-15/12-16
  buffered mode
    end-of-file marks 12-19
    errors, recovering from 12-85/12-86
    example of 12-19/12-20
    introduction to 12-16/12-17
    invoking 12-17/12-18
    revoking 12-18
    streaming devices 12-19
  closing 12-84
  communicating with 12-1/12-109
  errors, recovering from 12-52, 12-85/12-89
  forward spacing by files 12-5
  forward spacing by records 12-7/12-8
  I/O with 1-7, 2-23
  opening 12-4

Magnetic tape
  positioning 12-4/12-11
  reading, concepts of 12-11/12-13
  rewinding 12-10/12-11
  See also Labeled tape
  See also Unlabeled tape
  writing to, concepts of 12-13/12-14
Magnetic tape files 2-3
Main procedure
  TNS 17-4
  TNS/R native 17-9
Main stack 1-8, 16-8, 16-9, 17-6
  bounds checking 17-15/17-16
  sizing 17-13/17-15
  use for procedure calls 17-9
Main stack, TNS/E 16-9, 17-13
MAP CLASS DEFINEs
  example of 7-3/7-4
  FILE attribute 2-10
  purpose 2-10, 7-2
Matrix line printers, communicating with 11-2/11-9, 11-19/11-49
MBCS 1-9, 19-49/19-65
MBCS_CHARSIZE_ procedure 19-50, 19-54/19-55
MBCS_CHARSTRING_ procedure 19-51
MBCS_CHAR_ procedure 19-50, 19-52/19-53
MBCS_CODESETS_SUPPORTED_ procedure 19-50, 19-51/19-52
MBCS_DEFAULTCHARSET_ procedure 19-50, 19-52
MBCS_EXTERNAL_TO_TANDEM_ procedure 19-51
MBCS_FORMAT_CRT_FIELD_ procedure 19-51
MBCS_FORMAT_ITI_BUFFER_ procedure 19-51
MBCS_REPLACEBLANK_ procedure 19-50, 19-54
MBCS_SHIFTSTRING_ procedure 19-50, 19-55
MBCS_TANDEM_TO_EXTERNAL_ procedure 19-51
MBCS_TESTBYTE_ procedure 19-55/19-56
MBCS_TRIMFRAGMENT_ procedure 19-50, 19-54
MEDIACOM utility 12-21
Memory pools
  debugging 17-48
  defining 17-43/17-45
  getting information about 17-48
  getting space in 17-46/17-47
  introduction to 17-42/17-43
  resizing 17-47/17-48
  returning space to 17-47
Memory stack, TNS/E 16-9, 17-13
Memory, virtual 16-4, 16-5
Message system 1-6
Message tag 6-21
MESSAGESTATUS procedure 6-3, 6-25
Metacharacters
  See File-name patterns
  See Wild-card characters
Mirrored disk 1-3
Modifiers, formatter 19-6
Mom process 16-12
Monitor 1-15
Monitor process 1-2, 1-15
MONITORCPUS procedure 22-7/22-8
Monitoring a child process 16-46/16-47
MONITORNET procedure 22-7/22-8
Monolithic program
  See Unified program
MOUNTMSG attribute 12-28
MOVEX procedure 17-1, 17-37/17-38
multibyte 1-9
Multibyte character sets (MBCS) 1-9, 19-49/19-65

Multibyte characters
    analyzing strings of 19-52/19-53
    blank characters 19-54
    case shifting 19-55
    checking support for 19-51/19-52
    determining the default set 19-52
    example program 19-56/19-65
    introduction to 19-49/19-51
    size of 19-54/19-55
    special symbols, testing for 19-55/19-56
    trimming fragments of 19-54
Multithreaded server 22-1/22-2
MYPROCESSTIME procedure 18-19

N

Named processes
    creating 2-14
    relationship with 16-11/16-12
Native Inspect 25-2
    introduction 25-5
    specifying the debugging environment 25-6
    use in native envrionments 25-5
Native processes 1-23, 16-5
NEC Kanji character set 19-50
NetBatch utility 16-13
Next-record pointer 2-18/2-23
nld utility 25-7
Node down message 22-7/22-8
Node names 2-5
    scanning 13-5
    validating 13-5
Nonrepeatable edit descriptors 19-6
Nonretryable errors 2-28
NonStop Kernel Open System Services (OSS) Glossary-19
NonStop SQL files 1-5, 2-1
NonStop Transaction Manager/MP (TM/MP) 1-2, 1-8
Nowait I/O 1-5, 4-1/4-23
    example 4-12/4-23
    introduction to 4-1/4-3
    multiple files 4-10/4-11, 15-42/15-45
    multiple I/Os with one file 4-5/4-9
    on SIO files 15-39/15-45
    one I/O with one file 4-3/4-4, 15-39/15-41
NO^ERROR procedure 15-2
NUMBEREDIT procedure 14-1, 14-15/14-16
Numeric strings
    conversion to binary numbers 19-35/19-36
    obtaining from binary numbers 19-36
NUMIN procedure 19-1, 19-35/19-36
NUMOUT procedure 19-1, 19-36

O

Object file format 1-22
OLDFILENAME_TO_FILENAME_ procedure 8-4, 10-4/10-5
One-and-a-half-way communication 2-25
One-way communication
    example of 6-15/6-16, 15-49/15-51
    introduction to 2-25
    opening $RECEIVE 6-15
    receiving messages 6-15, 15-48/15-51
    sending messages 15-48/15-51
Open message 6-26, 6-29, 15-53, 22-5/22-6
OPENEDIT_ procedure 14-1, 14-8/14-10
Opener table
    adding a requester 22-5/22-6
    checking a request against 22-6
    contents of 22-3/22-4
    deleting a requester 22-6/22-8
    maintaining 22-3/22-8
    when CPU fails 22-7/22-8
    when requester closes server 22-7

OPENER_LOST_ procedure 22-7/22-8

OPEN^FILE procedure
  ABORT^OPENERR flag 15-61/15-62
  ABORT^XFERERR flag 15-62/15-63
  block buffering 15-25/15-26
  creating and opening files 15-24/15-25
  EDIT files 15-38
  flag values, setting 15-23/15-24
  interactive read prompt, setting 15-29
  introduction to 15-2
  long writes 15-30/15-31
  MUSTBENEW flag 15-25
  nowait I/O 15-40
  opening files 15-24
  padding records before writing 15-33/15-34
  printing 15-36
  purging data 15-26
  redirecting error messages 15-61
  suppressing error message reporting 15-60
  trimming trailing blanks before writing 15-32/15-33
  trimming trailing blanks on reading 15-34/15-35

OSS signal
  defined Glossary-21

OUT file 9-3
  and the SYSTEM command 8-2
  contents of 8-4
  opening 10-4/10-5
  with SIO 15-7/15-10

Overstriking text 11-27/11-29

OWNER attribute 12-28

P

P register 17-4

PACKEDIT procedure 14-1, 14-13/14-15

Page mode
  interrupt characters 10-20/10-22
  overview 10-18/10-19
  page-termination character 10-19
  pseudopolled terminals 10-22/10-23
  setting 10-9/10-10

Page-termination character 10-19

PAID 16-10/16-11

PARAM 1-7

PARAM command 8-2

Param message
  example 8-11
  introduction to 8-8/8-11
  processing 8-14/8-15
  receiving 8-12, 16-46
  sending 16-46
  structure of 8-11

Partitioned files
  creating 5-77/5-80
  introduction to 5-77
  opening 5-80

Pascal, calling Guardian procedures from 1-20

Passive backup 27-29

Passwords, controlling from $CMON
  local passwords 23-14/23-15
  remote passwords 23-15/23-17

Password^msg message 23-14/23-15

Password^reply structure 23-15

Pathway 1-8

pause function 25-24

PCB 16-2

PCL 11-5/11-29, 15-37
  selecting for 5577 printer 11-10/11-11

Permanent disk files
  creating 2-11
  naming 2-4/2-6

Persistent process 1-2, 1-15

PEXTDECS file 1-20

PFS, sizing 16-32/16-33

Physical-block length, SIO 15-18/15-19

PIN 2-8, 16-3

POOL_CHECK_ procedure 17-1, 17-48

POOL_DEFINE_ procedure 17-1, 17-43/17-45

POOL_GETINFO_ procedure 17-1, 17-48

POOL_GETSPACE_ procedure 17-1, 17-46/17-47

POOL_PUTSPACE_ procedure 17-1, 17-47

POOL_RESIZE_ procedure 17-1, 17-47/17-48

POSITION procedure 2-21, 2-22, 5-17, 5-32/5-33

POSITIONEDIT procedure 14-1, 14-12

PostScript
- selecting for 5577 printer 11-10/11-11

Pound sign
- in qualifier
  - device name 2-7
  - process name 2-8
- in temporary file ID 2-6

Prelogon^msg message 23-9/23-10

Prelogon^reply structure 23-9/23-10

Primary extent 2-12

Primary key 5-7

Printer control language (PCL) 11-5/11-29, 15-37

Printer files 2-3

Printers
- accessing 11-2/11-5
- closing 11-4/11-5
- communicating with 11-1/11-49, 15-36/15-37
- controlling 11-6/11-8
- escape sequences 11-6/11-9
- I/O with 1-7, 2-23
- opening 2-17, 11-4/11-5
- opening for SIO 15-36

Printing 11-1/11-49
- default printer settings 11-19, 11-29
- duplex mode 11-12
- example program for 11-31/11-49
- forms control 11-21/11-25
- job control commands 11-11/11-14
- number of copies 11-11
- overstriking text 11-27/11-29
- page control commands 11-14/11-16, 11-20/11-21
- page length 11-20
- page size 11-14, 11-15
- paper source 11-12
- setting margins 11-15, 11-20
- simplex mode 11-12
- underlining text 11-18, 11-27

Priv stack 1-8, 16-8, 16-9, 17-6, 17-14
- use of 17-9

Procedure call stack
- See User data stack

Process
- definition of 16-1/16-2
- startup 1-7

Process access ID 16-10/16-11

Process control block (PCB) 16-2

Process deletion message 16-37/16-38, 16-47/16-52

Process descriptor
- extracting subparts of 13-17/13-18
- introduction to 16-2
- syntax of 2-8

Process file names
- converting to process handles 16-62/16-63
- introduction to 16-2/16-3
- named processes 2-7
- obtaining from process handles 16-63/16-64
- process descriptors 2-8
- searching for 13-24

Process file names (continued)
  unnamed processes 2-8
Process file segment 17-6
Process file segment (PFS) sizing 16-32/16-33
Process files
  introduction to 2-4
  opening 2-17/2-18
Process handles
  converting to process file names 16-63/16-64
  converting to process strings 16-64
  getting information from 16-62
  introduction to 16-3
  manipulating 16-61/16-64
  obtaining from process file names 16-62/16-63
Process identification number (PIN) 2-8, 16-3
Process identifiers 16-2/16-3
Process names
  See Process file names
Process pairs 1-2, 1-15, 16-11
Process pairs, in fault-tolerant programming 27-1/27-2
Process priority 16-15/16-17, 23-22/23-24
Process qualifiers 2-8
Process sequence number 2-7, 2-8
Process subtype, introduction to 16-15
Process time
  canceling timers 18-17/18-18
  converting to readable form 18-19
  definition of 18-15
  setting timers 18-17/18-18
Process time signal message 18-17/18-18
Processcreation^accept structure 23-19
Processcreation^msg message 23-17/23-22
Processcreation^reject structure 23-19/23-20
Processes
  activating 16-52, 16-53
  code spaces 16-5/16-7
  communicating with 6-1/6-63, 15-45/15-52
  creating 2-13/2-15, 16-19/16-38
    controlling creation from $CMON 23-17/23-22
    errors 16-27/16-28
    introduction to 16-19/16-21
    named process 2-14, 16-23/16-25
    nowait 16-26/16-27
    specifying attributes and resources 16-28/16-38
    unnamed process 2-15, 16-23
  data spaces 16-8/16-10
  Debug state 16-18/16-19, 25-1/25-9
  deleting
    abnormal deletion 16-49
    completion codes 16-49/16-50
    introduction to 16-47/16-52
    other process 16-50/16-51
    own process 16-48/16-50
    using stop mode 16-51/16-52
  device subtype, specifying 16-35
  getting information about 16-53/16-58
  heap, sizing 17-13
  high PIN 16-28/16-31
  Inspect state 16-18/16-19
  I/O with 2-24/2-25, 6-1/6-63
    example of one-way communication 6-15/6-16
    example of one-way communication with SIO 15-49/15-52
    example of requester/server application 6-31/6-63
    example of two-way communication 6-13/6-14
    example of two-way communication with SIO 15-47

I/O with (continued)
  job ID, specifying 16-35/16-36
  low PIN 16-28/16-31
  main and priv stack, sizing 17-13
  managing 1-8, 16-1/16-64
  named
    creating 2-14, 16-23/16-25
    process file name for 2-7
  opening 2-17/2-18, 6-4/6-5
  organization of 16-4, 16-5
  persistent 1-15
  PFS, sizing 16-32/16-33
  priority of 16-15/16-17
  processor, specifying 16-35
  propagating DEFINEs 16-36/16-37
  receiving messages from 6-11/6-16, 15-46, 15-48/15-51
  replying to messages 6-11/6-16, 6-21, 15-46
  runnable state 16-18/16-19, 16-53
  saveabend state 16-18/16-19
  security of 16-10/16-11
  sending messages to 6-4/6-7
    no reply 6-6/6-7, 15-48
    reply expected 6-7, 15-46
  setting attributes
    nonstring 16-60/16-61
    string 16-61
  specifying attributes and resources 16-28/16-38
  starting state 16-18/16-19
  states of 16-18/16-19
  stop mode 16-11
  stopping state 16-18/16-19
  suspended state 16-18/16-19, 16-52/16-53
  suspending
    introduction to 16-52
    other process 16-52/16-53
    own process 16-52
  swap file, specifying 16-33/16-34
  timing 18-19
  TNS and TNS/R native 16-5
  TNS/E native 16-4
  unnamed
    creating 2-15, 16-23
    process file name for 2-8
  user data segment, sizing 16-32, 17-3
  user library file, specifying 16-33
PROCESSHANDLE_DECOMPOSE_ procedure 16-1, 16-62
PROCESSHANDLE_GETMINE_ procedure 16-60
PROCESSHANDLE_TO_FILENAME_ procedure 16-1, 16-63/16-64
PROCESSHANDLE_TO_STRING_ procedure 16-1, 16-64
PROCESSNAME_CREATE_ procedure 16-24/16-25
Processor, specifying 16-35, 23-33/23-35
PROCESSTIME procedure 18-19
PROCESS_ACTIVATE_ procedure 16-1, 16-53
PROCESS_CREATE_ completion message 16-22, 16-26/16-27
PROCESS_CREATE_ procedure 16-20
  creating processes
    detailed discussion 16-38
    introduction to 2-13/2-15
  Debug state 25-3/25-4
  memory management 17-1
  naming processes 2-7
  specifying debugging environment 25-8/25-9
PROCESS_DEBUG_ procedure 25-3

PROCESS_GETINFOLIST_ procedure
  introduction to 16-1
  multiple processes 16-57/16-58
  single process 16-56/16-57
PROCESS_GETINFO_ procedure
  getting the home terminal name 2-17, 9-3, 10-6
  how to use 16-54/16-56
  introduction to 16-1
  process timing information 18-19
  terminals 10-2
PROCESS_GETPAIRINFO_ procedure 16-58
  how to use 16-59
PROCESS_LAUNCH_ completion message 16-22, 16-26/16-27
PROCESS_LAUNCH_ procedure
  creating processes
    detailed discussion 16-19/16-20
  Debug state 25-4
  introduction to 16-1
  memory management 17-1
  specifying debugging environment 25-9
PROCESS_LAUNCH_PARMS_ definition 16-21
PROCESS_SETINFO_ procedure 13-1/13-2, 16-1, 16-60/16-61
PROCESS_SETSTRINGINFO_ procedure 16-1, 16-61
PROCESS_SPAWN_ procedure 16-20
PROCESS_STOP_ procedure 16-1, 16-47/16-52
PROCESS_SUSPEND_ procedure 16-1, 16-52/16-53
Program
  debugging 1-9, 25-1/25-9
  definition of 16-1/16-2
  monolithic
    See Program, unified
  unified 1-10
Program counter register 17-4
Programs
  native, TNS, and accelerated 16-3
Pseudopolled terminals 10-22/10-23
pTAL, calling Guardian procedures from 1-18
Public library
  DLL 16-4, 16-5
PURGE command 5-14

Q

Question mark (?) 13-3/13-4
Queue files 5-8

R

ra register 17-9
RAISE_ procedure/C function 25-24
Random access 2-21
READ procedure
  devices 2-23
  effect on record pointers 2-18
  entry-sequenced files 5-32/5-33
  introduction to 2-18
  key-sequenced files 5-49
  magnetic tape 12-2, 12-11/12-13
  processes 6-3
  relative files 5-17
  terminals 10-2, 10-6
  $RECEIVE 6-3, 6-15
READEDIT procedure 14-1, 14-12, 14-13
READEDITP procedure 14-1, 14-13
READLOCKX procedure 17-35/17-36
READUPDATE procedure
  effect on record pointers 2-22
  key-sequenced files 5-49/5-51
  relative files 5-17
  $RECEIVE 2-25
    concurrent message processing 6-18
    introduction 6-3
    system messages 6-27

- $RECEIVE (continued)
  - two-way communication 6-11
- READUPDATEX procedure
  - effect on record pointers 2-22
  - extended data segments 17-35/17-36
  - key-sequenced files 5-49/5-51
  - relative files 5-17
  - $RECEIVE 2-24, 2-25
    - concurrent message processing 6-18
    - introduction 6-3
    - system messages 6-27
    - two-way communication 6-11
- READX procedure
  - devices 2-23
  - effect on record pointers 2-18/2-22
  - entry-sequenced files 5-32/5-33
  - extended data segments 17-35/17-36
  - introduction to 2-18
  - key-sequenced files 5-49/5-51
  - magnetic tape 12-2, 12-11/12-13
  - processes 6-3
  - relative files 5-17
  - terminals 10-2, 10-6
  - $RECEIVE 6-3, 6-15
- Read-only access, files 3-1
- Read/write access, files 3-1
- READ^FILE procedure
  - interactive read prompt, changing 15-29
  - introduction to 15-2
  - nowait I/O 15-40/15-41
  - reading records 15-28
  - receiving interprocess messages 15-46, 15-48/15-49
  - sending interprocess messages 15-46
  - system messages 15-54
- Receive depth 2-25, 6-10, 6-17
- RECFORM attribute 12-27
- RECLEN attribute 12-26
- Record blocking 12-15/12-16, 12-26/12-27, 12-72
- Record length, magnetic tape 12-15/12-16
- Record length, SIO 15-67/15-71
- Reduced instruction-set computing (RISC) Glossary-24
- REELS attribute 12-24, 12-50
- Relative files
  - altering attributes of 5-18
  - alternate key access, example 5-70/5-71, 5-86/5-88
  - cache flushes 5-18
  - closing 5-18
  - creating 5-16/5-17
  - example of programming with 5-18/5-30
  - introduction to 2-2
  - I/O with 5-17
  - locking 5-18
  - opening 5-17
  - positioning 5-17
  - purging 5-18
  - purpose of 5-2
  - renaming 5-18
  - structure of 5-2/5-4
- Remote CPU down message 15-53, 22-7/22-8
- Remotepassword^msg message 23-15/23-17
- Remotepassword^reply structure 23-16/23-17
- Repeatable edit descriptors 19-6
- Reply 1-6
- Reply count 6-20
- REPLY procedure
  - introduction to 6-3
  - multiple concurrent messages 6-21
  - one-and-a-half-way communication 2-25

REPLYX procedure
  extended data segments 17-35/17-36
  introduction to 6-3
  multiple concurrent messages 6-21
  one-and-a-half-way communication 2-25
  two-way communication 2-24, 6-11/6-12
Requester process 1-10/1-15
  application control 21-3/21-4
  data mapping 21-2/21-3
  example 6-32/6-51, 21-9/21-51
  field validation 21-2
  functions of 21-1/21-4
  selecting a server 21-3/21-4
  terminal interface 21-2
  transaction control 21-4
  writing a program for 21-1/21-51
Requester/server model 1-10/1-15
  adding functions to 1-12
  adding users to 1-11
  fault tolerant 1-15
  in network environment 1-14/1-15
  load balancing 1-14
  modularity 1-11
  monitors 1-15
Request-for-device-type-information message 24-12/24-13
Resetsync message 6-26, 6-30
RESIZESEGMENT procedure 17-1, 17-34/17-35
Response
  See reply
RETENTION attribute 12-27
Retryable errors 2-28
RISC Glossary-25
RISC instructions Glossary-25
RUCB
  introduction to 15-5
  setting up 15-6
RUN command 8-2
  name-option parameter 2-7, 2-13
  RUNNAMED flag 2-13
RUN INSPECT command 25-8
RUND command 25-4/25-5
Runnable state 16-18/16-19, 16-53
RUNV command 25-4
Run-unit control block (RUCB)
  introduction to 15-5
  setting up 15-6

S

SAFEGUARD program 3-3
Saveabend state 16-18/16-19
SAVEABEND TAL compiler directive 25-7
SEARCH CLASS DEFINEs
  example of 7-4/7-5
  purpose 7-2
Secondary extents 2-12
Security, file 3-2/3-3
SEGMENT_ALLOCATE_ procedure
  default use 17-23/17-27
  extensible data segment 17-27
  introduction to 17-1
  read-only data segment 17-27
  shared data segment 17-27
  swap file, specifying 17-26/17-27
SEGMENT_DEALLOCATE_ procedure 17-1, 17-42
SEGMENT_GETINFO_ procedure 17-2, 17-34
  checking segment type 17-28
SEGMENT_USE_ procedure 17-2, 17-28/17-29
Selectable segments
  allocating 17-24
  introduction to 17-17/17-22
  making current 17-28/17-29
  referencing data in 17-30/17-31
Sequential access, unstructured file 2-19

Sequential Input/Output (SIO)
  See SIO
Server process 1-10/1-15
  concurrent message processing 6-17/6-22
  context-free 22-3
  example 6-51/6-63, 22-8/22-64
  functions of 22-1/22-3
  monitor 1-15
  monitor process 1-15
  multithreaded 22-1/22-2
  opener table
    See Opener table
  processing system messages 22-4/22-8
  single message handling 6-10/6-16
  single-threaded 22-1/22-2
  writing a program for 22-1/22-64
ServerWare Storage Management Foundation (SMF) 2-9
SET INSPECT command 25-7/25-8
SETJMP_ procedure/C function 25-24, 25-26, 25-32
SETLOOPTIMER procedure/C function 25-27
SETMODE
  function-4 3-10
Setmode message 6-26, 6-30, 24-10/24-11
SETMODE procedure
  device functions 9-4
  function-10 10-3
  function-11 10-3, 10-31
  function-110 10-3
  function-113 10-3
  function-119 12-3, 12-19, 12-28, 12-73
  function-12 10-3, 10-30/10-31
  function-120 12-3
  function-13 10-3
  function-14 10-3, 10-15
  function-141 17-36/17-37
  function-146 5-35

SETMODE procedure (continued)
  function-149 5-85
  function-152 5-13
  function-162 12-3
  function-20 10-3, 10-9
  function-22 10-3, 11-3
  function-23 10-3
  function-24 10-3
  function-25 11-3
  function-26 11-3
  function-260 11-3, 11-10
  function-27 10-3, 11-3
  function-28 10-3, 11-3
  function-29 11-3
  function-30 4-5/4-8
  function-36 6-9
  function-37 11-3
  function-38 10-3
  function-4 3-6/3-7
  function-5 11-3
  function-52 12-3, 12-13
  function-6 10-3, 10-16/10-17, 11-3, 11-28
  function-66 12-3, 12-72/12-73
  function-67 10-3
  function-68 11-3
  function-7 10-3, 10-11
  function-72 4-8/4-9
  function-8 10-3, 10-9/10-10
  function-80 6-25, 24-4/24-5
  function-9 10-3, 10-14/10-15, 10-21/10-22
  function-92 2-13
  function-99 12-3, 12-17/12-19, 12-29, 12-74
  interprocess communication functions 6-3
  magnetic tape functions 12-3
  memory management 17-2
  printer functions 11-3

SETMODE procedure (continued)
  terminal functions 10-2, 10-3
SETMODENOWAIT procedure
  device functions 9-4
  interprocess communication 6-3
  magnetic tape functions 12-2
  printer functions 11-2
  terminal functions 10-2
Setparam message 6-26, 6-30, 24-11
SETPARAM procedure
  device functions 9-4
  function-3 10-25/10-27, 10-30, 10-31
  interprocess communication 6-3
  terminal functions 10-2
SETSTOP procedure 16-1, 16-51/16-52
SETSYSTEMCLOCK procedure 18-27/18-28
SETTIME command 18-28
Setting timers
  elapsed time 18-16/18-17
  process time 18-17/18-18
SET^BREAKHIT operation 15-59/15-60
SET^CRLF^BREAK operation 15-55/15-56
SET^EDITREAD^POSITION operation 15-38/15-39
SET^ERRORFILE operation 15-61
SET^FILE procedure 15-16, 15-17, 15-67
  ASSIGN^BLOCKLENGTH operation 15-18/15-19
  ASSIGN^FILECODE operation 15-17
  ASSIGN^LOGICALFILENAME operation 15-10
  ASSIGN^OPENACCESS operation 15-15/15-16, 15-38
  ASSIGN^PRIMARYEXTENTSIZE operation 15-18
  ASSIGN^SECONDARYEXTENTSIZE operation 15-18
  differences between native and TNS 15-3
  INIT^FILEFCB operation 15-67
SET^FILE procedure (continued)
  INIT^FILEFCB^D00 15-67
  introduction to 15-2, 15-15
  SET^BREAKHIT operation 15-59/15-60
  SET^CRLF^BREAK operation 15-55/15-56
  SET^EDITREAD^POSITION operation 15-38/15-39
  SET^ERRORFILE operation 15-61
  SET^PRINT^ERR^MSG operation 15-60/15-61
  SET^PROMPT operation 15-29
  SET^READ^TRIM operation 15-34/15-35
  SET^SYSTEMMESSAGESMANY operation 15-53/15-54
  SET^WRITE^FOLD operation 15-30/15-31
  SET^WRITE^PAD operation 15-33/15-34
  SET^WRITE^TRIM operation 15-32/15-33
  SET^XFERERR operation 15-63
SET^PRINT^ERR^MSG operation 15-60/15-61
SET^PROMPT operation 15-29
SET^READ^TRIM operation 15-34/15-35
SET^SYSTEMMESSAGESMANY operation 15-53/15-54
SET^WRITE^FOLD operation 15-30/15-31
SET^WRITE^PAD operation 15-33/15-34
SET^WRITE^TRIM operation 15-32/15-33
SET^XFERERR operation 15-63
Shared Run-Time Library
  See SRL
Shared run-time library (SRL) 16-5, 16-5/16-7
SHIFTSTRING procedure 19-1, 19-37, 19-50, 19-55
SIGACTION_ procedure/C function 25-24, 25-26
SIGACTION_INIT_ procedure 1-10

SIGACTION_INIT_ procedure/C function 25-27/25-36

SIGACTION_RESTORE_ procedure/C function 25-27/25-36

SIGACTION_SUPPLANT_ procedure/C function 25-27/25-36

SIGADDSET_ procedure/C function 25-24, 25-26

SIGDELSET_ procedure/C function 25-24, 25-26

SIGEMPTYSET_ procedure/C function 25-24, 25-26

SIGFILLSET_ procedure/C function 25-24, 25-26

SIGISMEMBER_ procedure/C function 25-24

SIGJMP_MASKSET_ procedure/C function 25-27/25-36

SIGLONGJMP_ procedure/C function 25-25, 25-26, 25-29/25-36

Signal
- defined Glossary-27

Signal handler
- default 25-23
- defined Glossary-27
- examples of 25-29/25-36
- introduction to 25-23

Signal handling 1-9, 25-20/25-36

SIGNALPROCESSTIMEOUT procedure 18-17/18-18

Signals
- comparing traps and 25-22
- deferrable and nondeferrable 25-22
- interoperability in TNS and native processes 25-29
- introduction to 25-21/25-22
- standard POSIX.1 25-23/25-27
- Tandem extensions 25-27/25-36

SIGNALTIMEOUT procedure 18-16/18-17
- measuring long intervals 18-20/18-24

SIGNAL_ procedure/C function 25-25

SIGPENDING_ procedure/C function 25-25, 25-26

SIGPROCMASK_ procedure/C function 25-25, 25-26

SIGSETJMP_ procedure/C function 25-25, 25-26, 25-29/25-36

SIGSUSPEND_ procedure/C function 25-25

Singlethreaded server 22-1/22-2

SIO 1-7, 15-1/15-83
- BREAK handling 15-54/15-60
- example initialization
  - using TAL or pTAL DEFINEs 15-19/15-22
  - without TAL or pTAL DEFINEs 15-68/15-71
- example program 15-71/15-83
- file control blocks (FCBs)
  - allocating 15-7/15-10, 15-66
  - initializing 15-7/15-10, 15-67
  - introduction to 15-2
  - naming 15-7/15-10, 15-67
- file initialization
  - compile time 15-5/15-22
  - dynamic 15-65/15-71
  - run time 15-65/15-71
  - static 15-5/15-22
  - using TAL or pTAL DEFINEs 15-5/15-22
  - without TAL or pTAL DEFINEs 15-65/15-71
- interprocess messages 15-45/15-52
- introduction to 15-1/15-3
- logical file names, assigning 15-10
- multiple processes 15-52
- naming FCBs 15-7/15-10, 15-67
- one-way communication, example of 15-49/15-51
- PCL commands 15-37
- physical file name, reassigning 15-19
- printer control 15-37
- receiving interprocess messages 15-46, 15-48/15-51

SIO (continued)
  replying to interprocess messages 15-46
  sending interprocess messages
    no reply 15-48
    reply expected 15-46
  system messages 15-52/15-54
  two-way communication, example of 15-47
  writing a program for 15-3
  $RECEIVE file, using for 15-7/15-10, 15-66, 15-67, 15-67
SIO errors
  fatal 15-61/15-63
  introduction to 15-60
  I/O errors 15-62/15-63
  open errors 15-61/15-62
  redirecting 15-61
  retryable 15-63/15-65
  suppressing error-message reporting 15-60/15-61
SIO files
  access mode 15-15/15-16, 15-67/15-71
  ASSIGN command with 15-14/15-19
  AWAITIO procedure with 15-42/15-45
  block buffering 15-25/15-26
  block length 15-67/15-71
  closing 15-65
  creating and opening 15-24/15-25
  exclusion mode 15-16, 15-67/15-71
  extents 15-18, 15-67/15-71
  file code 15-17, 15-67/15-71
  flag values, setting 15-23/15-24
  getting information about 15-26/15-27
SIO files (continued)
    initializing
    compile time 15-5/15-22
    dynamic 15-65/15-71
    run time 15-65/15-71
    static 15-5/15-22
    using TAL or pTAL DEFINEs 15-5/15-22
    without TAL or pTAL DEFINEs 15-65/15-71
  interactive read prompt, changing 15-29
  I/O with 15-27/15-37
  logical-record length 15-17
  long writes to 15-30/15-31
  nowait I/O 15-39/15-45
  opening 15-24
  padding blank characters before writing 15-33/15-34
  physical-block length 15-19
  purging data 15-26
  reading 15-28
  read-only access 15-28
  read/write access 15-28
  record length 15-67/15-71
  trimming trailing blanks before writing 15-32/15-33
  trimming trailing blanks on reading 15-34/15-35
  write-only access 15-28
  writing 15-29
sleep C function 25-25
SMF
  See ServerWare Storage Management Foundation
Snapshot files, debugging
  TNS 25-1
  TNS/E 25-1, 25-2
  TNS/R 25-1
sp register 17-9

SRL data segments 17-6
SRL (shared run-time library) 16-5, 16-5/16-7
Stack frames 17-9
Stack pointer 17-9
Stacks
  TNS/R and TNS/E 16-9, 17-13
Stack, TNS user data 17-2
Starting state 16-18/16-19
Startup message
  example 8-5
  example of sending and receiving 16-39/16-46
  introduction to 8-3/8-5
  opening terminal files 10-4/10-5
  processing 8-6/8-8
  reading 8-6
  structure of 8-4
Startup sequence
  introduction to 1-7, 2-10, 8-2/8-3
  sending and receiving 16-38/16-46
  timing out 8-16
  with INITIALIZER 8-3/8-15
  without INITIALIZER 8-16/8-18
Start/stop mode devices
  labeled tape 12-28
  unlabeled tape 12-73
State information, updating 27-11/27-23
Stop mode 16-11, 16-51/16-52
Stopping state 16-18/16-19
Streaming devices
  labeled tape 12-28
  unlabeled tape 12-73
Strings
  case shifting 19-36/19-37
  converting between characters and numbers 19-35/19-36
  downshifting 19-37
  editing 19-37/19-46
  sorting 19-46/19-49
Strings (continued)
  upshifting 19-37
Strings, numeric
  conversion to binary numbers 19-35/19-36
  obtaining from binary numbers 19-36
STRING_UPSHIFT_ procedure 19-55
Structured files 2-2
Subvolume names
  resolving 13-8/13-9
  scanning 13-6
  validating 13-6
Subvolumes 2-2
Suspended state 16-18/16-19, 16-52/16-53
Swap file 16-33/16-34, 17-26/17-27
Swap space guarantee
  requesting from KMSF 16-34/16-35
Sync depth 2-16, 27-6, 27-15
Synchronizing processes 1-24, 26-1
SYSTEM attribute 12-24
System clock
  checking 18-26
  setting 18-27/18-28
System code 16-5/16-7
SYSTEM command 8-2
System library 16-4, 16-5, 16-5/16-7
System messages
  introduction to handling 6-26
  masking 15-53/15-54
  reading, in SIO 15-54
  receiving 6-27
  selecting 15-53/15-54
  with SIO 15-52/15-54
  -100 (Remote CPU down) 15-53, 22-7/22-8
  -101 (Process deletion) 16-37/16-38, 16-47/16-52
  -102 (PROCESS_CREATE_ completion) 16-22, 16-26/16-27
  -102 (PROCESS_LAUNCH_ completion) 16-22, 16-26/16-27

System messages (continued)
  -103 (Open) 6-26, 6-29, 15-53, 22-5/22-6
  -104 (Close) 6-26, 6-29, 15-53, 22-7
  -105 (Break-on-device) 24-15
  -110 (Loss of communication with a network node) 15-53
  -110 (Node down) 22-7/22-8
  -2 (CPU down) 15-53, 22-7/22-8
  -22 (Time signal) 18-16/18-17
  -26 (Process time signal) 18-17/18-18
  -32 (Control) 6-26, 6-30, 24-10
  -33 (Setmode) 6-26, 6-30, 24-10/24-11
  -34 (Resetsync) 6-26, 6-30
  -35 (Controlbuf) 6-26, 6-30
  -37 (Setparam) 6-26, 6-30, 24-11
  -40 (Request for device-type information) 24-12/24-13
System time
  introduction to 18-1/18-2
  managing 18-25/18-36

T

TACL
  communicating with 8-1/8-21
  displaying text 8-20/8-21
  obtaining startup information from 8-3/8-8
  See also $CMON
  setting up process environment 8-1/8-3
  startup without INITIALIZER 8-16/8-18
  using ASSIGNs 8-8/8-15
  using PARAMs 8-8/8-15
  waking 8-19/8-20
TAKE^BREAK procedure 15-2, 15-55/15-56
TAL, calling Guardian procedures from 1-18
Tandem Advanced Command Language (TACL)
  See TACL
TAPE CLASS DEFINEs
  BLOCKLEN attribute 12-26
  CLASS attribute 12-23, 12-70
  DENSITY attribute 12-27, 12-72
  DEVICE attribute 12-27, 12-71
  EBCDIC attribute 12-25
  example of 7-5
  EXPIRATION attribute 12-27
  FILEID attribute 12-24
  FILESECT attribute 12-49
  FILESEQ attribute 12-24
  GEN attribute 12-28
  LABELS attribute 12-23, 12-70
  MOUNTMSG attribute 12-28
  OWNER attribute 12-28
  purpose 7-3
  RECFORM attribute 12-27
  RECLEN attribute 12-26
  REELS attribute 12-50
  RETENTION attribute 12-27
  SYSTEM attribute 12-24
  USE attribute 12-25
  VERSION attribute 12-28
  VOLUME attribute 12-24
Tape headers 12-80
TAPECOM utility 12-21
Temporary disk files
  creating 2-12
  naming 2-6
Terminal files 2-3
Terminal simulator
  BREAK key 24-13/24-15
  device subtype 24-2
  introduction to 24-1
  I/O requests 24-5/24-9
  naming 24-3

Terminal simulator (continued)
  system messages 24-3/24-5, 24-9/24-13
Terminals
  accessing 10-1/10-10
  block mode
    See Page mode
  BREAK key 10-23/10-36
  BREAK mode 10-27/10-36
  BREAK ownership 15-55/15-59
    releasing 10-26
    taking 10-25/10-26
  closing 10-10
  communicating with 10-1/10-41
  conversational mode
    See Conversational mode
  echo mode 10-9
  errors 10-36/10-41
  form feed 10-17/10-18
  interrupt characters
    See Interrupt characters
  I/O with 1-6, 2-23, 10-6/10-8
  line spacing 10-16/10-17
  line-termination character 10-11/10-12
  opening 10-4/10-6
  page mode
    See Page mode
  pseudopolled 10-22/10-23
  reading from 10-6
  simulating
    See Terminal simulator
  timeout 10-8
  transfer mode 10-9/10-10
  writing to 10-6
  writing to and reading from 10-7/10-8
TIME command 18-26
TIME procedure 18-26
Time signal message 18-16/18-17
Time zones 18-2
Timekeeping 1-9
TIMESTAMP procedure 18-3, 18-14
Timestamps
  See 48-bit timestamps
  See Julian timestamps
Time, managing 1-9, 18-1/18-36
Timing, intervals 1-9, 18-7/18-8, 18-15/18-19
  long 18-20/18-24
TM/MP 1-2, 1-8
TNS processes 1-23, 16-5
TNS/E debugging tools 25-1
TNS/R execution modes 1-23
TNS/R native processes 1-23, 16-5
TNS/R native signal
  defined Glossary-33
TNS/R systems
  trap handling on 25-15
To-RISC shell 16-5/16-7
Transaction Manager/MP (TM/MP) 1-2, 1-8
Transparent mode 10-15
Trap
  defined Glossary-33
Trap handler
  default 25-10/25-11
  defined Glossary-34
  examples of 25-16/25-20
  exiting 25-13/25-15
  introduction to 25-9/25-11
  setting up 25-11/25-12
  trap processing 25-12/25-13
Trap handling 1-9, 25-9
  on TNS/R systems 25-15
TS_NANOSECS_ procedure 18-4
TS_UNIQUE_COMPARE_ procedure 18-4
TS_UNIQUE_CONVERT_ procedure 18-4
TS_UNIQUE_CONVERT_TO_JULIAN_ procedure 18-4
TS_UNIQUE_CREATE_ procedure 18-2

Two-way communication
  example of 6-13, 6-31/6-63, 15-47
  introduction to 2-24, 6-10
  opening $RECEIVE 6-10
  receiving messages 6-11, 15-46
  replying to messages 6-11/6-12, 15-46
  returning data 6-11
  returning errors 6-11
  sending messages 15-46

U

Underlining text 11-18, 11-27
Unified program 1-10
Unlabeled tape
  accessing 12-70/12-74
  appending to last file 12-78/12-79
  appending to only file 12-76/12-77
  backward spacing by files 12-5/12-10
  backward spacing by records 12-8/12-10
  blocking records 12-72
  buffered mode 12-74
  checking for end of tape 12-80/12-81
  closing 12-84
  conversion mode, selecting 12-73/12-74
  device mode, selecting 12-73
  device name for 12-71
  errors, recovering from 12-52, 12-85/12-89
  example program 12-89/12-109
  forward spacing by files 12-5
  forward spacing by records 12-7/12-8
  introduction to 12-1, 12-70
  label type, specifying 12-70
  opening 12-4
  positioning 12-4/12-11
  reading from multiple-file tape 12-83/12-84
  reading from multiple-volume file 12-84
Unlabeled tape (continued)
  reading from only file 12-82/12-83
  reading, concepts of 12-11/12-13
  rewinding 12-10/12-11
  tape density, specifying 12-72/12-73
  tape headers 12-80
  writing files to multiple-file tape 12-77/12-78
  writing to multiple-volume file 12-79/12-81
  writing to scratch tape 12-75/12-76
  writing to, concepts of 12-13/12-14
UNLOCKFILE procedure 3-6/3-11
UNLOCKREC procedure 5-11
Unnamed processes
  creating 2-15
  relationship with 16-12
UNPACKEDIT procedure 14-1
Unstructured files
  altering attributes of 5-15
  cache flushes 5-13
  closing 5-13
  creating 5-9/5-10
  introduction to 2-2, 5-1, 5-9
  I/O with 2-18/2-23, 5-11
  locking 5-11
  opening 5-11
  positioning 2-18/2-23, 5-11
  purging 5-14
  random access 2-21
  reading 2-18/2-23, 5-11
  renaming 5-12
  sequential access 2-19
  writing 2-18/2-23, 5-11
USE attribute 12-25
User code 16-4, 16-5, 16-5/16-7
User data areas, managing 17-6

User data segment 1-8
- bounds checking 17-15, 17-15/17-16
- managing 17-2/17-15
- sizing 16-32, 17-3
- upper 64K bytes 17-6/17-15

User data stack 1-8, 16-8, 17-3/17-6
User library 16-5
- overview 16-4, 16-5

User library file 16-33
Users
- adding 23-24/23-26
- deleting 23-26/23-27

V

VERSION attribute 12-28
Vertical forms control (VFC) table 11-21/11-25
VFC table 11-21/11-25
Virtual memory 16-4, 16-5
- introduction to 1-8
- managing 1-8
- pools 1-9, 17-42/17-48

Visual Inspect
- introduction 25-1, 25-5
- specifying the debugging environment 25-6
- use in native envrionments 25-1, 25-5
- use with TNS/E native programs 25-1

VOLUME attribute 12-24
VOLUME command 8-2
Volume names 2-5
Volumes 2-2

W

Waited I/O 1-5
WAIT^FILE procedure 15-2
- multiple files 15-42/15-45
- one file 15-41

Wakeup message 8-19/8-20
WHO command 9-3
Wild-card characters 13-3/13-4
Working set
- checking for errors 7-13
- restoring 7-13, 7-16/7-17
- saving 7-16/7-17
- setting attributes in 7-12/7-13

WRITE procedure
- devices 2-23
- effect on record pointers 2-18
- entry-sequenced files 5-32/5-33
- introduction to 2-18
- key-sequenced files 5-49/5-51
- magnetic tape 12-2, 12-13/12-14
- printers 11-2
- relative files 5-17
- terminals 10-2
- writing messages 6-4

WRITEEDIT procedure 14-1, 14-12/14-13, 14-13/14-15
WRITEEDITP procedure 14-1, 14-13/14-15
WRITEREAD procedure
- terminals 2-23, 10-2
- writing messages 6-4, 6-7

WRITEREADX procedure 2-24, 6-7
- extended data segments 17-35/17-36
- terminals 2-23, 10-2, 10-7/10-8
- writing messages 6-4, 6-7

WRITEUPDATE procedure
- effect on record pointers 2-22
- key-sequenced files 5-49
- relative files 5-17

WRITEUPDATELOCKX procedure 17-35/17-36
WRITEUPDATEUNLOCKX procedure 17-35/17-36
WRITEUPDATEX procedure
- effect on record pointers 2-22/2-23
- extended data segments 17-35/17-36
- key-sequenced files 5-49/5-51
- relative files 5-17

- WRITEX procedure
  - devices 2-23
  - effect on record pointers 2-18/2-21
  - entry-sequenced files 5-32/5-33
  - extended data segments 17-35/17-36
  - introduction to 2-18
  - key-sequenced files 5-49/5-51
  - magnetic tape 12-2, 12-13/12-14
  - printers 11-2
  - relative files 5-17
  - terminals 10-2, 10-6
  - writing messages 6-4, 6-6/6-7
- Write-only access, files 3-1
- WRITE^FILE procedure 15-2
  - long writes 15-30/15-31
  - printers 15-37
  - replying to interprocess messages 15-46
  - sending messages, no reply 15-48
  - writing records 15-29

Z

- ZSYS* files
  - ZSYSC 1-24
  - ZSYSCOB 1-24
  - ZSYSDDL 1-23
  - ZSYSPAS 1-24
  - ZSYSTAL 1-24
- ZSYS^DDL^SMSG^PROCCREATE^DEF definition 16-22
- ZSYS^VAL^LEN^FILENAME literal 2-11, 2-12, 2-14
- ZSYS^VAL^LEN^PROCESSDESCR literal 2-14
- ZSYS^VAL^PCREATOPT^NAMEDBYSYS literal 2-15, 16-25
- ZSYS^VAL^PCREATOPT^NAMEINCALL literal 2-14, 16-23
- ZSYS^VAL^PCREATOPT^NONAME literal 2-15, 16-23

Special Characters

- # symbol
  - See Pound sign
- $ symbol
  - in device name 2-7
  - in process name 2-7
  - in volume name 2-5
- $CMON
  - adding users, controlling 23-24/23-26
  - command interface to
    - CPU designation 23-33/23-35
    - introduction to 23-27/23-29
    - logon display text, setting 23-29/23-31
    - refusing command interpreter requests 23-31/23-33
  - communicating with TACL 23-2/23-3
  - configuring TACL 23-4/23-8
  - CPU of new process, controlling 23-21/23-22
  - debugging 23-89/23-92
  - deleting users, controlling 23-26/23-27
  - example of 23-36/23-70
  - example of command interface to 23-70/23-88
  - illegal logon, controlling 23-13/23-14
  - introduction to 23-1/23-3
  - logoff, controlling 23-12
  - logon, controlling 23-9/23-11
  - passwords, controlling
    - local 23-14/23-15
    - remote 23-15/23-17
  - process creation, controlling 23-17/23-22
  - process priority of new process 23-20, 23-22/23-24
  - TACL illegal logon, controlling 23-13/23-14
  - testing 23-89/23-92

$RECEIVE
    closing 6-13
    getting message information 6-19/6-21
    introduction to 1-6, 1-11, 2-4, 2-24
    opening 6-10, 6-15, 6-17
    queuing messages on 6-8/6-9
    reading messages from 6-18
    SIO
        allocating 15-66
        initializing FCB for 15-7/15-10, 15-67
        naming 15-67
* symbol 13-3/13-4
<, condition-code-less-than (CCL) 2-27
=, condition-code-equal (CCE) 2-27
=_DEFAULTS DEFINE 2-5, 7-3, 7-5/7-6
>, condition-code-greater-than (CCG) 2-27
? symbol 13-3/13-4
\ symbol, in node name 2-5
__ns_backup_fopen C function 27-6
__ns_fget_file_open_state C function 27-6
__ns_fget_file_state C function 27-7
__ns_fopen_special C function 27-6
__ns_fset_file_state C function 27-7
__ns_start_backup C function 27-5