DOMAIN

Order No. 005506
Revision 00
Software Release 9.0

Programming with General System Calls

Programming With General System Calls
Update 1

Order No. 008858
Revision 00
Software Release 9.2

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Printed in U.S.A.

First Printing: March, 1986

Preface

Programming With General System Calls describes the general-purpose DOMAIN system calls you can use to perform services for your programs.

Audience

This manual is intended for programmers who write applications and wish to make use of the system calls provided by DOMAIN . Before using this manual, you should be familiar with programming concepts and terminology, and should also understand the DOMAIN implementation of the programming language you are using.

This manual describes how to use system calls to perform programming tasks, and makes extensive use of programming examples to clarify explanations. However, the manual does not provide complete reference information for each call that it demonstrates. For complete reference information, see the *DOMAIN System Call Reference* manual.

Organization of this Manual

This manual contains nine chapters:

- Chapter 1 describes the predefined data type scheme used with system calls, and provides necessary data type information for C and FORTRAN programmers.
- Chapter 2 describes how to handle errors and faults.
- Chapter 3 describes how to invoke programs and how to obtain process information.
- Chapter 4 describes how to perform I/O using the IOS manager.
- Chapter 5 describes how to program the Display Manager.
- Chapter 6 describes how to use system-defined eventcounts.
- Chapter 7 describes how to manipulate time.
- Chapter 8 describes a variable formatting package for Pascal programmers.
- Chapter 9 describes how to access DOMAIN object types using IOS calls.

This manual uses excerpts of Pascal programs to illustrate the narrative descriptions. Most excerpts begin with the name of the program from which they were taken. To see the C translation, find the corresponding program in Appendix A.

You can also view the programs on-line, as described in the next section.

On-Line Sample Programs

The programs from this manual are stored on-line, along with sample programs from other DOMAIN manuals. We include sample programs in Pascal and C. All programs in each language have been stored in master files (to conserve disk space). There is a master file for each language.

In order to access any of the on-line sample programs you must create one or more of the following links:

```
(For Pascal examples)  $ crl ~com/getpas /domain_examples/pascal_examples/getpas

(For C examples)       $ crl ~com/getcc  /domain_examples/cc_examples/getcc
```

To extract a sample program from one of the master files, all you have to do is execute one of the following programs:

```
(To get a Pascal program)     $ getpas

(To get a C program )         $ getcc
```

These programs prompt you for the name of the sample program and the pathname of the file to copy it to. Here is a demonstration:

```
$ getpas
Enter the name of the program you want to retrieve -- stream_sio_access
What file would you like to store the program in? --  sio1.pas

Done.
$
```

You can also enter the information on the command line in the following format:

```
$ getpas name_of_program_to_retrieve  name_of_output_file
```

For example, here is an alternate version of our earlier demonstration:

```
$ getpas stream_sio_access sio1.pas
```

GETPAS and GETCC warn you if you try to write over an existing file.

For a complete list of on-line DOMAIN programs in a particular language, enter one of the following commands:

```
(for Pascal)   $ getpas help
(for C)        $ getcc help
```

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following conventions:

UPPERCASE Uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.

lowercase	Lowercase words or characters in formats and command descriptions represent values that you must supply.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in format and command descriptions. In simple Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down the <CTRL> key while typing the character.
...	Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Suggested Reading Paths

Before you read this manual, you should be familiar with the following:

- *Getting Started With Your DOMAIN System.* This manual provides general information about using your node.

- *DOMAIN System Call Reference (Volumes 1 and 2).* These manuals give complete reference information on all DOMAIN system calls.

In addition, you should be familiar with the DOMAIN language manuals for your programming language.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference* manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same information on-line by typing:

$ HELP CRUCR <RETURN>

For documentation comments, a Reader's Response form is located at the back of each manual.

Contents

Chapter 1 Using DOMAIN Predefined Data Types **1-1**

1.1. Why DOMAIN Uses Predefined Data Types 1-1
1.2. How to Use Insert Files 1-1
1.3. How to Use Predefined Constants and Values 1-3
1.4. How to Use DOMAIN Predefined Data Types 1-4
1.4.1. Enumerated Types 1-5
1.4.2. Sets 1-5
1.4.3. Records 1-6
1.4.4. Variant Records 1-6
1.4.5. Arrays 1-7
1.4.5.1. Arrays of Records 1-7
1.5. Basic Data Types 1-9
1.6. How to Use Data Type Reference Material 1-9
1.6.1. Data Types Sections 1-10
1.6.2. System Call Descriptions 1-12
1.6.2.1. Parameter Descriptions 1-12
1.6.3. Error Sections 1-12
1.7. Data Type Information for FORTRAN Programmers 1-13
1.7.1. Boolean Type 1-13
1.7.2. Pointers 1-14
1.7.3. Enumerated Types 1-16
1.7.4. Sets 1-18
1.7.4.1. Setting Bits 1-18
1.7.4.2. Testing Bits 1-19
1.7.4.3. Emulating Large Sets 1-21
1.7.5. Records 1-22
1.7.6. Variant Records 1-24
1.7.7. Passing Parameters to System Calls 1-29
1.7.7.1. Passing Integer Parameters 1-29
1.7.7.2. Passing Integer Constants 1-30
1.8. Data Type Information for C Programmers 1-30
1.8.1. Boolean Type 1-30
1.8.2. Sets 1-30
1.8.2.1. Setting Bits 1-31
1.8.2.2. Testing Bits 1-32
1.8.2.3. Emulating Large Sets 1-34
1.8.3. Records 1-36
1.8.4. Variant Records 1-36
1.8.5. Passing Parameters to System Calls 1-37
1.8.5.1. Passing Character Arrays 1-37
1.8.5.2. Passing Integer Parameters 1-38
1.8.5.3. Passing Integer Constants 1-40

Chapter 2 How to Handle Errors and Faults **2-1**

2.1. System Calls, Insert Files, and Data Types 2-1
2.2. Status Structure 2-1
2.2.1. Accessing Fields of the Status Code with FORTRAN 2-2
2.3. Testing for Errors 2-3
2.4. Printing Error Messages 2-3
2.5. Standardized Error Reporting 2-4
2.6. Testing for Specific Errors 2-6
2.6.1. Setting a Severity Level 2-8
2.7. Faults 2-8
2.7.1. Synchronous Faults 2-11
2.7.2. Asynchronous Faults 2-11
2.8. Handling Faults with Clean-Up Handlers 2-12
2.8.1. Establishing a Clean-Up Handler 2-12
2.8.2. Releasing a Clean-Up Handler 2-14
2.8.3. Multiple Clean-Up Handlers 2-15
2.8.4. Exiting a Clean-Up Handler 2-15
2.8.4.1. Resignaling Passing the Fault Status 2-15
2.8.4.2. Resignaling Passing a Severity Level 2-16
2.8.4.3. Re-establishing the Handler and Returning to the Program 2-16
2.8.4.4. Returning to the Program 2-17
2.8.5. Handling Errors With Clean-Up Handlers 2-17
2.9. Handling Faults with Fault Handlers 2-21
2.9.1. Establishing a Fault Handler 2-21
2.9.1.1. Writing the Fault-Handling Function 2-21
2.9.1.2. Establishing the Function as a Handler 2-22
2.9.1.3. Setting Target Faults 2-23
2.9.1.4. Specifying Handler Types 2-24
2.10. Inhibiting Asynchronous Faults 2-25

Chapter 3 Managing Programs **3-1**

3.1. System Calls, Insert Files, and Data Types 3-1
3.2. Invoking External User Programs 3-1
3.2.1. Invoking a Program in Wait Mode 3-2
3.2.1.1. Setting Severity Levels 3-3
3.2.2. Invoking a Program in Default Mode 3-6
3.2.2.1. Waiting for a Child Process 3-7
3.2.3. Invoking a Program in Background Mode 3-11
3.3. Passing Arguments to Invoked Programs 3-18
3.4. Accessing Arguments from an Invoked Program 3-20
3.4.1. Accessing Arguments with PGM_$GET_ARG 3-20
3.4.2. Accessing Arguments with PGM_$GET_ARGS 3-21
3.5. Deleting Arguments 3-22
3.6. Passing Streams to an Invoked Program 3-24
3.7. Getting Process Information 3-27
3.7.1. Getting Information About Your Process 3-27
3.7.2. Getting Information About Other Processes 3-29

Chapter 4 Performing I/O with IOS Calls **4-1**

4.1. System Calls, Insert Files, and Data Types 4-1
4.2. Overview of the IOS Manager 4-2
4.2.1. Stream Connections 4-3
4.2.2. Stream IDs 4-3
4.2.3. Default Stream IDs 4-3
4.2.4. Stream Markers 4-4
4.2.5. IOS Calls for Manipulating Streams 4-5
4.3. Creating and Opening Objects 4-5
4.3.1. Specifying an Object's Type 4-6
4.3.2. Controlling how IOS Creates Objects 4-7
4.3.3. Creating a Backup Object 4-8
4.3.4. Creating Temporary Objects 4-9
4.3.5. Examples of Opening and Creating Objects 4-9
4.3.6. Controlling how IOS Opens Objects 4-11
4.3.7. Controlling a Stream's Access and Concurrency 4-12
4.3.8. Example of Controlling an Object's Access and Concurrency 4-15
4.4. Reading and Changing Object Attributes 4-16
4.4.1. Inquiring about and Changing Object Attributes 4-18
4.4.2. Example of Inquiring about and Changing Attributes 4-19
4.4.3. Example of Changing Attributes 4-21
4.4.4. Getting Additional Information about Objects and Directories 4-24
4.5. Closing and Deleting Objects 4-24
4.6. Writing to Objects 4-25
4.6.1. Example of Writing to Objects 4-25
4.7. Reading Objects 4-27
4.8. Reading Objects Sequentially 4-29
4.9. Performing Random Access 4-31
4.9.1. Nonkeyed Seeking 4-31
4.9.2. Keyed Seeking 4-32
4.9.3. Example of Using Seek Keys 4-32
4.10. Handling Record-Oriented Object Types 4-35
4.10.1. Writing Fixed-Length Records 4-36
4.10.2. Writing Variable-Length Records 4-38
4.10.3. Reading Fixed-Length Records with Seek Keys 4-41
4.10.4. Record Formats 4-44

Chapter 5 Using the Display Manager **5-1**

5.1. System Calls, Insert Files, and Data Types 5-1
5.2. Overview of the Display Manager 5-2
5.3. Starting Out 5-4
5.3.1. Creating a New Pad in a New Window 5-4
5.3.2. Creating a New Pad in a Window Pane 5-5
5.4. Creating Subsequent Pads in Window Panes 5-6
5.4.1. Creating Input Pads in Window Panes 5-7
5.4.2. Creating Transcript Pads in Window Panes 5-8
5.4.3. Creating Edit Pads in Window Panes 5-9
5.4.4. Creating Read-Only Edit Pads in Window Panes 5-10

5.4.5. Closing Windows and Window Panes 5-10
5.4.6. Sample Program: Creating and Closing Windows and Window Panes 5-11
5.5. Manipulating Windows 5-14
5.5.1. Specifying a Window Number with PAD_$INQ_WINDOWS 5-14
5.5.2. Getting Window Positions with PAD_$INQ_WINDOWS 5-14
5.5.3. Getting Position of Window Borders with PAD_$INQ_FULL_WINDOW 5-17
5.5.4. Changing How Windows Look 5-18
5.5.5. Inquiring About the User's Display and Keyboard 5-21
5.5.6. Specifying Character Fonts 5-23
5.5.7. Changing Scale Factors 5-25
5.5.8. Getting Current Scale Factors with PAD_$INQ_FONT 5-27
5.5.9. Sample Program: Creating a Window to Run a Clock 5-29
5.6. Using Icons 5-34
5.6.1. Creating an Icon 5-34
5.6.2. Positioning an Icon 5-36
5.6.3. Creating Your Own Icon Font 5-38
5.6.4. Sample Program: Using Icons 5-38
5.7. Handling Graphics Input with Frames 5-42
5.7.1. Creating the Frame 5-42
5.7.2. Clearing the Frame 5-43
5.7.3. Sample Program: Creating and Writing to Frames 5-43
5.8. Sending and Receiving Program Input 5-49
5.8.1. Processing System Input in Cooked Mode 5-49
5.8.2. Bypassing System Input Processing with Raw Mode 5-49
5.8.3. Controlling System Output with Cursors 5-52
5.8.4. Writing to an Output Stream: Control Codes and Escape Sequences 5-56
5.9. Using Paste Buffers 5-57
5.9.1. Reading and Writing to Paste Buffers 5-57
5.9.2. Sample Program: Using Paste Buffers 5-58
5.10. Using the Touchpad Manager 5-61
5.10.1. Absolute Mode 5-61
5.10.2. Relative Mode 5-62
5.10.3. Absolute/Relative Mode 5-62
5.10.4. Changing Touchpad Sensitivity with Scale Factors 5-63
5.10.5. Timing Factors for the Touchpad or Bitpad in Relative Mode 5-63
5.10.6. Changing the Origin in Absolute Mode with TPAD_$SET_MODE 5-63
5.10.7. Setting the Origin in Relative Mode with TPAD_$SET_CURSOR 5-64
5.10.8. Hysteresis Factor 5-64

Chapter 6 Using Eventcounts **6-1**

6.1. EC2 System Calls, Insert Files, and Data Types 6-1
6.2. Overview of Eventcounts 6-2
6.3. How the System Uses Eventcounts 6-4
6.4. Getting and Reading Eventcounts 6-5
6.5. Waiting for Events 6-7
6.6. Responding to Events and Incrementing the Trigger Value 6-9
6.7. Handling Asynchronous Faults during Eventcount Waits 6-13
6.7.1. Disabling Asynchronous Faults with EC2_$WAIT 6-14
6.7.2. Disabling Asynchronous Faults with EC2_$WAIT_SVC 6-16

Chapter 7 Manipulating Time **7-1**

7.1. CAL and TIME System Calls, Insert Files, and Data Types 7-1
7.2. How the System Represents Time 7-1
7.3. Getting System Time 7-2
7.3.1. Getting Local Time 7-2
7.3.2. Timezone Offsets 7-4
7.4. Converting from System Time to Readable Time 7-6
7.5. Converting from Readable Time to System Time 7-7
7.6. Manipulating Time 7-9
7.6.1. Relative Time 7-9
7.6.2. Adding Times 7-10
7.6.3. Subtracting Times 7-11
7.6.4. Comparing Times 7-13
7.7. Suspending Process Execution 7-16
7.8. Using the Time Eventcount 7-18

Chapter 8 Formatting Variables with VFMT **8-1**

8.1. VFMT System Calls, Insert Files, and Data Types 8-1
8.2. Data Types That Can Be Formatted with VFMT 8-2
8.3. Routine Syntax 8-2
8.4. Simple Examples 8-2
8.5. Building Control Strings 8-4
8.5.1. Format Directive Overview 8-4
8.5.2. Inserting Literal Text 8-5
8.5.3. Repeating Control Strings 8-5
8.6. Format Directive Usage 8-5
8.6.1. Formatting ASCII Data: The %A Directive 8-5
8.6.2. Formatting Floating Point Data: The %F and %E Directives 8-8
8.6.3. Formatting Integer Data: The %O, %D, and %H Directives 8-10
8.6.4. Special Control String Directives 8-12
8.6.5. Format-Related Directives 8-13
8.7. Examples 8-14
8.7.1. Building a Character Table 8-14
8.7.2. Parsing an Input Line 8-15
8.7.3. Reading Strings Using a Variety of Formats 8-19

Chapter 9 Accessing DOMAIN Types with IOS Calls **9-1**

9.1. Overview of DOMAIN Object Types 9-2
9.2. Accessing Mailboxes 9-3
9.2.1. Opening a Mailbox with IOS_$OPEN 9-4
9.2.2. Performing I/O on Mailboxes with IOS Calls 9-4
9.2.3. Example of Accessing a Mailbox with IOS Calls 9-5
9.3. Accessing Serial Lines 9-8
9.3.1. Opening a Stream to a Serial Line 9-8
9.3.2. Setting Serial Line Object Characteristics 9-8
9.3.3. Performing I/O across a Serial Line 9-9

9.3.4. Example of Accessing an SIO Line 9-10
9.4. Accessing Files on Magnetic Tape 9-12
9.4.1. Creating and Opening a Magtape Descriptor Object 9-12
9.4.2. Reading and Changing Magtape Descriptor Attributes 9-13
9.4.3. Closing a Magtape Descriptor Object 9-13
9.4.4. Example of Writing to a Magtape File 9-14
9.4.5. Example of Reading from a Magtape File 9-18

Appendix A Sample Programs in C **A-1**

A.1. PFM_CLEAN_UP.C A-6
A.2. PGM_SHELL.C A-8
A.3. PGM_INVOKE.C A-9
A.4. PGM_OPEN.C A-10
A.5. PGM_EC.C A-11
A.6. PGM_INVOKE_DIVIDE.C A-14
A.7. PGM_DIVIDE.C A-17
A.8. PGM_ZERO_HANDLER.C A-18
A.9. PGM_ORPHAN.C A-19
A.10. PGM_PASS_ARGS.C A-20
A.11. PGM_PASSEE_ARG.C A-22
A.12. PGM_PASSEE.C A-23
A.13. PGM_DEL_INV.C A-24
A.14. PGM_PASS_STREAMS.C A-26
A.15. PGM_YOUR_PROC.C A-28
A.16. PGM_CHILD_INFO.C A-30
A.17. STREAM_INQ_REC_LEN.C A-32
A.18. STREAM_CHANGE_EXP.C A-34
A.19. STREAM_PUT_FIXED.C A-37
A.20. STREAM_PUT_VAR.C A-39
A.21. STREAM_PUT_VAR_UASC.C A-41
A.22. STREAM_GET_VAR.C A-43
A.23. STREAM_GET_VAR_UASC.C A-45
A.24. STREAM_UPDATE.C A-48
A.25. STREAM_WRITE_TAPE.C A-52
A.26. STREAM_READ_TAPE.C A-56
A.27. STREAM_SIO_ACCESS.C A-60
A.28. STREAM_MBX_CLIENT.C A-62
A.29. STREAM_LIST_LINKS.C A-64
A.30. PAD_MAKE_WINDOWS.C A-66
A.31. PAD_INQ_WINDOW_SIZE.C A-69
A.32. PAD_FULL_WINDOW_SHOW.C A-71
A.33. PAD_WINDOW_SHOW.C A-73
A.34. PAD_INQ_DISP_KBD.C A-77
A.35. PAD_SCALE.C A-80
A.36. PAD_INQ_FONT.C A-83
A.37. PAD_DIGCLK.C A-85
A.38. PAD_MAKE_ICON.C A-89
A.39. PAD_CREATE_ICON.C A-92
A.40. PAD_FILENAME.C A-96
A.41. PAD_RAW_MODE.C A-102

A.42. PAD_PASTE_BUFFER.C A-105
A.43. EC_TIME_KBD_EVENTS.C A-108
A.44. EC_WAIT_FOR_TIME.C A-111
A.45. CAL_DECODE_LOCAL.C A-114
A.46. TIME_ZONE.C A-115
A.47. CAL_ADD_TIMES.C A-117
A.48. CAL_SUB_TIMES.C A-118
A.49. TIME_COMPARE.C A-120
A.50. TIME_WAIT_ABS.C A-123
A.51. TIME_WAIT_OR_DEFAULT.C A-125

Index **Index-1**

Illustrations

Figure 1-1.	The Pointer/Data Relationship	1-15
Figure 2-1.	The Structure of the Status Data Type	2-2
Figure 3-1.	Argument Vector/Argument Configuration	3-19
Figure 4-1.	Record-Oriented Object with Count Fields	4-45
Figure 4-2.	Record-Oriented Object without Count Fields	4-46
Figure 4-3.	Unstructured Record-Oriented Object	4-46
Figure 5-1.	The DEBUG Display with the -SRC Option	5-3
Figure 6-1.	Relationship Between a Process and an Eventcount	6-3

Tables

Table 1-1.	Summary of Insert Files	1-2
Table 2-1.	Summary of Faults	2-9
Table 2-2.	Synchronous Program Faults	2-11
Table 2-3.	Synchronous System Faults	2-11
Table 2-4.	Types of Fault Handlers	2-24
Table 3-1.	Severity Levels	3-4
Table 4-1.	Default Streams	4-4
Table 4-2.	IOS Calls to Manipulate Stream Connections	4-5
Table 4-3.	Object Types	4-7
Table 4-4.	Controlling IOS_$CREATE when a Name Refers to an Existing Object	4-8
Table 4-5.	Options That Control how to Open Streams	4-11
Table 4-6.	IOS Options for Specifying Access Types and Concurrency Modes	4-13
Table 4-7.	Access/Concurrency Combinations for Shared Streams	4-14
Table 4-8.	Object Attributes	4-16
Table 4-9.	FORTRAN Carriage Control Characters	4-16
Table 4-10.	Stream Connection Attributes	4-17
Table 4-11.	Type Manager Attributes	4-18
Table 4-12.	Getting Additional Information about an Object	4-24
Table 4-13.	Options to Control an IOS_$PUT call	4-25
Table 4-14.	Options to Control an IOS Get Call	4-28
Table 4-15.	Available Record Formats	4-45
Table 5-1.	PAD System Calls to Create and Manipulate Icons	5-34
Table 5-2.	Control Codes to Format Output to Windows and Panes	5-56
Table 5-3.	Escape Sequences	5-57
Table 5-4.	Touchpad Scale Factor Values for Display	5-63
Table 6-1.	Summary of EC2 System Calls	6-2
Table 6-2.	EC2 Calls for Obtaining Pointers to Eventcounts	6-5
Table 6-3.	Wait Actions When Asynchronous Faults are Enabled	6-13
Table 6-4.	Wait Actions When Asynchronous Faults are Inhibited	6-14
Table 6-5.	Program Results if a Fault Occurs During a Wait	6-18
Table 8-1.	Summary of Format Directives	8-6
Table 8-2.	%A: Format ASCII Data	8-7
Table 8-3.	%F and %E: Format Floating Point Data	8-9
Table 8-4.	%O, %D, and %H: Format Integer Data	8-11
Table 9-1.	Default SIO Descriptor Objects Pathnames	9-8
Table A-1.	Summary of C Programs in Appendix A	A-1

Examples

Example 2-1.	A Simple Error-Handling Procedure	2-3
Example 2-2.	Formatting Error Messages with System Calls	2-5
Example 2-3.	Testing for Specific STREAM Errors	2-7
Example 2-4.	Establishing A Clean-Up Handler	2-13
Example 2-5.	Invoking a Clean-Up Handler for an Error	2-19
Example 2-6.	Writing a Fault-Handling Function	2-22
Example 2-7.	Establishing a Fault Handler	2-24
Example 3-1.	Invoking an Existing Shell Command	3-3
Example 3-2.	Returning a Severity Level from an Invoked Program	3-5
Example 3-3.	Using an Eventcount to Wait for a Child Process	3-8
Example 3-4.	Invoking a Program in Background Mode	3-13
Example 3-5.	Converting a Child Process to an Orphan Process	3-17
Example 3-6.	Passing Arguments to an Invoked Program	3-19
Example 3-7.	Accessing Arguments with PGM_$GET_ARG	3-21
Example 3-8.	Accessing Arguments with PGM_$GET_ARGS	3-22
Example 3-9.	Deleting an Argument from the Argument Vector	3-23
Example 3-10.	Passing Streams to an Invoked Process	3-25
Example 3-11.	Getting Information About Your Process	3-28
Example 3-12.	Getting Information About an Invoked Process	3-30
Example 4-1.	Creating an Object	4-9
Example 4-2.	Opening an Existing Object	4-10
Example 4-3.	Checking for Compatible Access Type and Concurrency Modes	4-15
Example 4-4.	Inquiring About an Object	4-19
Example 4-5.	Changing an Object from RIW to Write Access	4-21
Example 4-6.	Writing to a UASC Object Line by Line	4-26
Example 4-7.	Reading Sequentially From an Object	4-29
Example 4-8.	Accessing a UASC Object Randomly Using Seek Keys	4-33
Example 4-9.	Writing Fixed-Length Records	4-36
Example 4-10.	Writing Variable-Length Records	4-39
Example 4-11.	Seeking Fixed-Length Records	4-41
Example 5-1.	Creating a New Pad with PAD_$CREATE_WINDOW	5-5
Example 5-2.	Creating a New Pad with PAD_$CREATE	5-5
Example 5-3.	Creating an Input Pad in a Window Pane	5-7
Example 5-4.	Creating a Transcript Pad in a Window Pane	5-8
Example 5-5.	Creating an Edit Pad in a Window Pane	5-9
Example 5-6.	Creating and Closing Windows and Window Panes	5-11
Example 5-7.	Getting Size and Position of Windows	5-15
Example 5-8.	Using PAD Calls to Manipulate a Full Window	5-17
Example 5-9.	Changing How a Window Looks	5-19
Example 5-10.	Inquiring About User's Display and Keyboard	5-21
Example 5-11.	Selecting a Character Font File	5-24
Example 5-12.	Setting Scale Factors to Raster Units with PAD_$SET_SCALE	5-26

Example 5-13. Using PAD_$INQ_FONT 5-28
Example 5-14. Using PAD Calls to Create a Clock 5-30
Example 5-15. Changing a Window to an Icon 5-35
Example 5-16. Creating an Icon 5-35
Example 5-17. Changing Icon Position and Character 5-36
Example 5-18. Using Icons 5-39
Example 5-19. Creating a Frame 5-42
Example 5-20. Displaying a Filename at the Top of a File 5-44
Example 5-21. Using Raw Mode 5-50
Example 5-22. Using PAD_$CPR_ENABLE to Report Cursor Positions 5-53
Example 5-23. Using Paste Buffers 5-58
Example 6-1. Getting and Reading System-Defined Eventcounts 6-6
Example 6-2. Waiting for System-Defined Eventcounts 6-8
Example 6-3. Responding to System-Defined Eventcounts 6-10
Example 6-4. Handling Asynchronous Faults with A Time Eventcount 6-15
Example 6-5. Handling Asynchronous Faults with EC2_$WAIT_SVC 6-17
Example 7-1. Getting Local Time Using an Offset 7-3
Example 7-2. Getting Local Time in Readable Format 7-3
Example 7-3. Getting Timezone Offset and Name 7-5
Example 7-4. Converting from System Format to Readable Format 7-7
Example 7-5. Converting Time from ASCII strings to System Format 7-8
Example 7-6. Adding a Relative Time to an Absolute Time 7-10
Example 7-7. Subtracting Two Times 7-11
Example 7-8. Comparing Two File Creation Times 7-13
Example 7-9. Suspending Process Execution for a Relative Time 7-16
Example 7-10. Suspending Process Execution Until an Absolute Time 7-17
Example 7-11. Using a Time Eventcount to Repeat a Prompt 7-19
Example 8-1. Writing (Encoding) a Variable to Output using VFMT_$WRITE 8-3
Example 8-2. Decoding a Variable using VFMT_$READ 8-3
Example 8-3. Building a Character Table of ASCII Characters 8-14
Example 8-4. Parsing an Input Line 8-15
Example 8-5. Reading Strings Using a Variety of VFMT Formats 8-20
Example 9-1. Writing to and Reading from a Mailbox 9-5
Example 9-2. Accessing a Serial Line 9-10
Example 9-3. Writing to a Magtape File 9-15
Example 9-4. Reading from a Magtape File 9-19

Chapter 1
Using DOMAIN Predefined Data Types

DOMAIN provides predefined data types, constants, and values to make using the DOMAIN system calls easier. This chapter describes how to use these predefined types and values. It includes sections that address the special needs of FORTRAN and C programmers.

1.1. Why DOMAIN Uses Predefined Data Types

The DOMAIN system provides predefined data types to use when calling system routines to facilitate passing arguments between your program and the system. Using a predefined data type lets you declare in a single line of code a complex data type that would otherwise require a lengthy declaration.

Predefined data types are especially useful when using a programming language that supports user-defined data types; C and Pascal are two such languages.

FORTRAN, however, does not support user-defined data types. A FORTRAN programmer must declare each data type using standard FORTRAN data types. This makes the declaration of some DOMAIN data types more involved for FORTRAN programmers. For this reason, Section 1.7 describes in detail how FORTRAN programmers should declare each DOMAIN data type.

1.2. How to Use Insert Files

The DOMAIN system routines are divided, by function, into several subsystems. The routines of each subsystem are prefixed for easy indentification. A subsystem prefix consists of a number of identifying characters followed by the special characters "_$". For example, the routines that perform stream functions are prefixed with STREAM_$. These subsystem prefixes are also used to distinguish DOMAIN data types and constants that are used by the subsystem routines.

The DOMAIN predefined data types for each subsystem are declared in a separate file, known as an insert file. When you use a routine belonging to a certain subsystem, you must include that subsystem's corresponding insert file. For some languages, the insert files define the required number and type of each system call parameter.

Insert files are located in the directory /SYS/INS/. There is one insert file per subsystem for each programming language. Include the appropriate insert file for your programming language. For example, if you are using error routines in a Pascal program, you include the insert file /SYS/INS/ERROR.INS.PAS. Using the same routines in a FORTRAN program, you include /SYS/INS/ERROR.INS.FTN. All insert files are specified using the syntax:

/SYS/INS/subsystem-prefix.INS.language-abbreviation

where language abbreviation is PAS (Pascal), FTN (FORTRAN), or C (C). Table 1-1 shows a list of all the available insert files.

In addition to including required subsystem insert files in a program, you must always include the BASE insert file for your programming language. When specifying more than one insert file, the BASE insert file should be specified first.

BASE insert files are specified using the syntax:

/SYS/INS/BASE.INS.language-abbreviation

These files contain some basic definitions that are used by a number of subsystem routines. See Section 1.5 for details about the BASE file.

Table 1-1. Summary of Insert Files

Insert File	Operating System Component
/SYS/INS/BASE.INS.lan	Base definitions -- must always be included
/SYS/INS/ACLM.INS.lan	Access control list manager
/SYS/INS/CAL.INS.lan	Calendar
/SYS/INS/ERROR.INS.lan	Error reporting
/SYS/INS/EC2.INS.lan	Eventcount
/SYS/INS/FAULT.INS.lan	Fault status codes
/SYS/INS/GM.INS.lan	Graphics Metafiles Resource
/SYS/INS/GMF.INS.lan	Graphics Map Files
/SYS/INS/GPR.INS.lan	Graphics Primitives
/SYS/INS/IPC.INS.lan	Interprocess communication datagrams
/SYS/INS/KBD.INS.lan	[Useful constants for keyboard keys]
/SYS/INS/MBX.INS.lan	Mailbox manager
/SYS/INS/MS.INS.lan	Mapping server
/SYS/INS/MTS.INS.lan	Magtape/streams interface
/SYS/INS/MUTEX.INS.lan	Mutual exclusion lock manager
/SYS/INS/NAME.INS.lan	Naming server
/SYS/INS/PAD.INS.lan	Display Manager
/SYS/INS/PBUFS.INS.lan	Paste buffer manager
/SYS/INS/PFM.INS.lan	Process fault manager
/SYS/INS/PGM.INS.lan	Program manager
/SYS/INS/PM.INS.lan	User process routines
/SYS/INS/PROC1.INS.PAS	Process manager (Pascal only)
/SYS/INS/PROC2.INS.lan	User process manager
/SYS/INS/RWS.INS.lan	Read/write storage manager
/SYS/INS/SIO.INS.lan	Serial I/O
/SYS/INS/SMDU.INS.lan	Display driver
/SYS/INS/STREAMS.INS.lan	Stream manager
/SYS/INS/TIME.INS.lan	Time
/SYS/INS/TONE.lan	Speaker
/SYS/INS/TPAD.INS.lan	Touchpad manager
/SYS/INS/VEC.INS.lan	Vector arithmetic
/SYS/INS/VFMT.INS.lan	Variable formatter

The suffix ".lan" varies with the language you're using; it is either ".FTN", ".PAS", or ".C".

In some cases, you may find insert files to be a useful on-line reference. Be aware, though, that the way in which insert files are written is not completely consistent. For complete and consistent information, use the *DOMAIN System Call Reference* manual.

1.3. How to Use Predefined Constants and Values

In addition to predefined data types, DOMAIN provides predefined values and constants that are used when calling system routines. The insert files define the values of all predefined constants, such as completion status codes.

Predefined values correspond to specific predefined data types. That is, if you have declared a variable to be of a certain predefined data type (an enumerated type or a set, see Section 1.4), then the values that the variable can have are limited to a number of predefined values. (However, not all predefined data types have predefined values.)

For example, in the third parameter of the PAD_$CREATE_WINDOW call, you must specify which type of pad you are creating. The predefined data type of the parameter is PAD_$TYPE_T (INTEGER*2 for FORTRAN). You can specify one of three predefined values, PAD_$EDIT, PAD_$READ_EDIT, PAD_$TRANSCRIPT. Of course, the program must include the PAD insert file to reference the PAD routines, data types, and values.

```
%include 'sys/ins/base.ins.pas';
%include 'sys/ins/pad.ins.pas';
%include 'sys/ins/streams.ins.pas';

 VAR
     { Delare variables. }
     type         : pad_$type_t;
     display_unit : integer;
     window       : pad_$window_desc_t;
     stream_win   : stream_$id_t;
     status       : status_$t;

 BEGIN
     { Load window values. }
              .
              .
              .
     { Load the parameter with predefined value. }
     type := pad_$transcript;

     display_unit := 1;

     pad_$create_window(' ',          { Null pathname for transcript pad }
                        0,            { Null namelength for transcript pad }
                        type,         { Type of pad }
                        display_unit, { Number of unit}
                        window,       { pad_$window_desc_t }
                        stream_win,   { stream ID of the new window }
                        status);      { Completion status }
```

You can specify predefined constants for *input* parameters in a call directly; you do not need to declare a variable to hold them. (The same is true for non-predefined constants, although, in this case, the call must expect a scalar type.) However, you must declare a variable to hold output and input/output parameters. The example above may be written:

```
%include 'sys/ins/base.ins.pas';
%include 'sys/ins/pad.ins.pas';
%include 'sys/ins/streams.ins.pas';

 VAR
    { Delare variables. }
    window        : pad_$window_desc_t;
    stream_win    : stream_$id_t;
    status        : status_$t;

 BEGIN
    { Load window values. }
             .
             .
             .
    pad_$create_window(' ',                { Null pathname for transcript pad }
                       0,                  { Null namelength for transcript pad }
                       pad_$transcript,    { Type of pad }
                       1,                  { Number of unit}
                       window,             { pad_$window_desc_t }
                       stream_win,         { stream ID of the new window }
                       status);            { Completion status }
```

The Data Types sections of the *DOMAIN System Call Reference* manual list any predefined values that a data type may have.

Note that although FORTRAN programs cannot use predefined data types, they can reference predefined constants and values. See Section 1.7 for details.

1.4. How to Use DOMAIN Predefined Data Types

Because the DOMAIN operating system is predominantly written in Pascal, the predefined data types reflect the data types available in that language. The following sections describe the different kinds of predefined data types, in Pascal terms. Each section contains the following information:

- The purpose of the data type.
- How to recognize the data type in the insert files.
- A program segment showing how to declare and load a variable of that data type.

The fact that C also supports user-defined data types permits C programs to use the predefined data types. For this reason, C programmers should find this section useful. However, C and Pascal are not completely compatible; some data type differences exist, and certain circumstances require C programmers to employ special programming techniques. Section 1.8 describes these differences and techniques in order to make programming on DOMAIN easier for C programmers. If you are a C programmer, read Section 1.8 before reading this section.

FORTRAN does not support user-defined data types. A FORTRAN programmer must declare all data types using standard FORTRAN data type statements. Section 1.7 describes how FORTRAN programmers should declare each DOMAIN data type.

1.4.1. Enumerated Types

Enumerated types are used by DOMAIN when an argument may contain one of a number of constant values. For example, the following is the insert file data type declaration of the mapped segment (MS) access mode parameter:

```
ms_$acc_mode_t =
                 (ms_$r,     { Read }
                  ms_$rx,    { Read and execute }
                  ms_$wr,    { Read and write }
                  ms_$wrx,   { Read, write, and execute }
                  ms_$riw);  { Read with intent to write }
```

The following program segment declares and loads a parameter of this type, in Pascal:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ms.ins.pas';

VAR
  { Declare parameter. }
  access  :  ms_$acc_mode_t;
           .
           .
           .
BEGIN
   { Load parameter with predefined value. }
   access := ms_$r;
```

1.4.2. Sets

A set type is used by DOMAIN when an argument can contain a combination of constant values. For example, the following is the insert file data type declaration of the Process Fault Manager (PFM) options parameter.

```
pfm_$fh_opt_set_t = SET OF  (pfm_$fh_backstop,
                             pfm_$fh_multi_level);
```

The following program segment declares and loads a parameter of this type, in Pascal:

```
%include '/sys/ins/pfm.ins.pas';

VAR
   { Declare parameter. }
   options          : pfm_$fh_opt_set_t;
           .
           .
           .
BEGIN
   { Load parameter with both predefined values. }
   options := [pfm_$fh_backstop, pfm_$fh_multi_level];
```

1.4.3. Records

A record type is used by DOMAIN when an argument contains multiple pieces of information that may be accessed separately.

For example, the following is the insert file data type declaration of the calendar (CAL) readable time format.

```
cal_$timedate_rec_t = PACKED RECORD   { Returned from cal_$decode_time }
    year:     integer ;
    month:    integer ;
    day:      integer ;
    hour:     integer ;
    minute:   integer ;
    second:   integer
    END ;
```

The following program segment declares and loads a parameter of this type, then accesses one field in it:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';

VAR
    d_clock : cal_$timedate_rec_t;
             .
             .
             .
BEGIN

    { Get decoded local time -- load d_clock. }
    cal_$decode_local_time (d_clock);

    { Access the year. }
    writeln ('the year is ',d_clock.year);
```

1.4.4. Variant Records

A variant record type is used by DOMAIN when an argument contains multiple pieces of information that may be typed differently, depending on usage.

For example, the following is the insert file data type declaration of the status parameter.

```
TYPE   status_$t = PACKED RECORD CASE integer OF
           1:  (fail:  boolean;        { TRUE if module couldn't handle error }
                subsys: 0..127;        { Subsystem code }
                modc: 0..255;          { Module code }
                code: integer);        { Module specific error }
           2:  (all: integer32);       { Used for testing for specific value }
           END ;
```

The following program segment declares and loads a parameter of this type, in Pascal:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
   status : status_$t;

BEGIN

   open(a_file_variable,
        file_name,
        'NEW',
        status);  { Returns status in one form }


   IF status.all <> status_$ok THEN
      writeln ('STATUS CODE IS :', status.code) { Writes it in another }
```

1.4.5. Arrays

An array type is used by DOMAIN when an argument contains a large number of smaller, identical data types. That is, an array of characters, an array of pointers, etc. The most commonly encountered array is the character array.

For example, the following is the insert file data type declaration of the pathname data type:

```
CONST  name_$pnamlen_max = 256;         { Max length of pathname }

TYPE   name_$pname_t = ARRAY [1..name_$pnamlen_max] OF char;
```

To declare and load the pathname parameter in Pascal, write:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/name.ins.pas';

VAR
  pathname : name_$pname_t;
           .
           .
           .
BEGIN

   writeln ('Input File Name: ');
   readln (pathname);
```

1.4.5.1. Arrays of Records

Arrays of records are used by DOMAIN when an argument contains a number of record structures. The graphics interface to DOMAIN (the GPR and GM subsystems) uses arrays of records to pass information.

One of the more complicated data types is the GPR_$WINDOW_LIST_T data type. It is an array of GPR_$WINDOW_T records. A GPR_$WINDOW_T record is made up of two fields that are, in turn, records made up of two fields.

The following is the insert file data type declaration of the GPR_WINDOW_LIST_T and all the declarations that make up a window record:

```
{ Lists of windows }
gpr_$window_list_t = ARRAY[1..10] OF gpr_$window_t;

{ Windows on a bitmap }
gpr_$window_t = RECORD
    window_base: gpr_$position_t;
    window_size: gpr_$offset_t
    END;

{ Bitmap positions }
gpr_$position_t = RECORD
    x_coord, y_coord: gpr_$coordinate_t
    END;

{ Bitmap offsets }
gpr_$offset_t = RECORD
    x_size, y_size: gpr_$coordinate_t;
    END;

{ Bitmap coordinates }
gpr_$coordinate_t = integer16;
```

The following program declares a window list and loads it by calling GPR_$INQ_VIS_LIST. It then writes the coordinates of the returned windows to standard output.

```
%include 'sys/ins/base.ins.pas';
%include 'sys/ins/gpr.ins.pas';

VAR
     num_of_windows   : integer;            { Number of subwin to return }
     total_windows    : integer;            { Number of subwin that exist }
     visible_list     : gpr_$window_list_t; { List of visible subwindows }

BEGIN

     num_of_windows := 2
     { Returns list of visible subwindows when a window is obscured. }
     gpr_$inq_vis_list( num_of_windows, { Number of subwindows to return }
                        total_windows,  { Returns number of subwin that exist }
                        visible_list,   { Returns list of visible subwindows }
                        status);

     { Print the visible window coordinates. }
     writeln ('VISIBLE WINDOW COORDINATES');

     n = 1;
```

```
DO WHILE (n <= num_of_windows) BEGIN
    WITH visible_list[n] DO
     writeln (n);
     writeln ('x-coordinate', window_base.x_coord);
     writeln ('y-coordinate', window_base.y_coord);
     writeln ('length', window_size.x_size);
     writeln ('height', window_size.y_size);
     n = n + 1;
     writeln ();
   END;
```

1.5. Basic Data Types

There are a number of data types that are used by more than one subsystem. They are defined in the BASE insert file. These data types include:

STATUS_$T	Describes a status code. The value of the status code tells whether a system call succeeded or failed. A detailed description of how to use the return status appears in Chapter 2.
NAME_$PNAME_T	Describes a DOMAIN pathname. A pathname is used to specify a system object.
STREAM_$ID	Describes a unique identifier for an I/O connection. The stream ID is used in most I/O system calls.
TIME_$CLOCK_T	Describes the internal clock representation of time.
UID_$T	Describes the unique identifier for a file type.

C programmers, note that the C BASE insert file predeclares a Boolean type to be an unsigned character type, and also declares a "true" and "false" value to test Booleans. See Section 1.8.1 for more information.

1.6. How to Use Data Type Reference Material

In addition to this task-oriented handbook, DOMAIN provides you with the *DOMAIN System Call Reference* manual. The reference is arranged alphabetically. The subsystems are ordered alphabetically, and each call within a subsystem is ordered alphabetically.

The material for each subsystem is organized into the following three parts:

1. Detailed data type information (including illustrations of records for the use of FORTRAN programmers).

2. Full descriptions of each system call.

3. List of possible error messages.

1.6.1. Data Types Sections

A subsystem's Data Types section precedes the subsystem's individual call descriptions. Each Data Types section describes the predefined constants and data types for a subsystem. These descriptions include an atomic data type translation (i.e., TIME_$REL_ABS_T = 2-byte integer) for use by FORTRAN programmers, as well as a brief description of the type's purpose. Where applicable, any predefined values associated with the type are listed and described. Below is an example of a data type description for the TIME_$REL_ABS_T type.

`TIME_$REL_ABS_T`	`A 2-byte integer. Indicator of type of time. One of the following predefined values:`
	
	`TIME_$RELATIVE - relative time`
	
	`TIME_$ABSOLUTE - absolute time`

In addition, the record data types are illustrated in detail. These illustrations are primarily intended to assist FORTRAN programmers in constructing record-like structures, but have been designed to convey as much information as possible for all programmers. Each record type illustration:

- Clearly shows FORTRAN programmers the structure of the record that they must construct using standard FORTRAN data type statements. The illustrations show the size and type of each field. (How to declare predefined records using FORTRAN is described in Section 1.7.)

- Describes the fields that make up the record.

- Lists the byte offsets for each field. These offsets are used to access fields individually.

- Indicates whether any fields of the record are, in turn, predefined records.

The following is the description and illustration of the CAL_$TIMEDATE_REC_T predefined record:

CAL_$TIMEDATE_REC_T

Readable time format. The diagram below illustrates the CAL_$TIMEDATE_REC_T data type:

predefined type	byte: offset		field name
	0:	integer	year
	2:	integer	month
	4:	integer	day
	6:	integer	hour
	8:	integer	minute
	10:	integer	second

Field Description:

year
Integer representing the year.

month
Integer representing the month.

day
Integer representing the day.

hour
Integer representing the hour (24 hr. format).

minute
Integer representing the minute.

second
Integer representing the second.

FORTRAN programmers, note that a Pascal variant record is a record structure that may be interpreted differently depending on usage. In the case of variant records, as many illustrations will appear as are necessary to show the number of interpretations. See Section 1.7.6 for details on how to handle variant records.

1.6.2. System Call Descriptions

The system call descriptions are listed alphabetically for quick reference. Each system call description contains:

- An abstract of the call's function.
- The order of call parameters.
- A brief description of each parameter.
- A description of the call's function and use.

These descriptions are standardized to make referencing the material as quick as possible.

1.6.2.1. Parameter Descriptions

Each parameter description begins with a phrase describing the parameter. If the parameter can be declared using a predefined data type, the descriptive phrase is followed by the phrase ",in XXX format", where XXX is the predefined data type. Pascal or C programmers, look for this phrase to determine how to declare a parameter.

FORTRAN programmers, use the second sentence of each parameter description for the same purpose. It describes the data type in atomic terms that you can use, such as, "This is a 2-byte integer". In complex cases, FORTRAN programmers are referenced to the respective subsystem's data type section. FORTRAN programmers should read Section 1.7 to learn how to construct complex DOMAIN data types in FORTRAN.

The rest of a parameter description describes the use of the parameter and the values it may hold.

The following is an example of a parameter description:

access

New access mode, in MS_$ACC_MODE_T format. This is a 2-byte integer. Specify only one of the following predefined values:

MS_$R	Read access.
MS_$WR	Read and write access.
MS_$RIW	Read with intent to write.

An object which is locked MS_$RIW may not be changed to MS_$R.

1.6.3. Error Sections

Each error section lists the status codes that may be returned by subsystem calls. The following information appears for each error:

- Predefined constant for the status code.
- Text associated with the error.

The following is a portion of the NAME Error Section:

NAME_$DIRECTORY_FULL	The directory has no room for any more objects.
NAME_$ALREADY_EXISTS	The pathname given is not unique.
NAME_$BAD_PATHNAME	The pathhname given is not a valid pathname.

See Chapter 2 for details on how to use status codes.

1.7. Data Type Information for FORTRAN Programmers

As stated above, DOMAIN predefined data types reflect the data types available in Pascal. FORTRAN programmers must emulate these data types using standard FORTRAN data type statements. You do not need to know Pascal to emulate these data types, but understanding the purpose of the Pascal data types is useful.

The following sections are organized by the data type to be emulated. Each section explains:

- The purpose of the data type.
- How to recognize the type in the reference material.
- How to emulate the type using FORTRAN.
- How to reference a variable of this type.

1.7.1. Boolean Type

Boolean types are variables that evaluate to either TRUE or FALSE. A Boolean value is described in the reference material and the insert files as a Boolean.

There are two ways to emulate a Boolean type in FORTRAN. Which way you use depends on the way the Boolean is used by the system. DOMAIN uses a Boolean either as a separate data type or as a field in a record structure.

If the system uses a Boolean as a separate type, emulate the Boolean type by using the LOGICAL type. A Pascal Boolean is one byte long and a LOGICAL is four bytes long. However, they both evaluate to TRUE or FALSE and a Boolean value returned from the system may be loaded into a logical parameter.

The following program segment declares a LOGICAL variable into which the system loads a Boolean value. The program then writes the value to output, using logical formatting.

```
*     Declare SIO_$ variables
      INTEGER*4    status
      INTEGER*2    stream_id
      LOGICAL      value_b  { Boolean value }
         .
         .
         .
*     INQUIRE  CTS_ENABLE
      CALL sio_$inquire (stream_id,
     2                  sio_$cts_enable,  { Option }
     2                  value_b,          { Returned by system }
     2                  status)

      IF (status .NE. status_$ok) THEN
          CALL error_$print (status)
      ENDIF

*     Print whether Boolean is TRUE (T) or FALSE (F)
      write (*,40) value_b
40    format ('The CTS_ENABLE is ',L5)
```

If the system uses a Boolean as a field in a record structure, declare the field to be a CHAR type. Although the fact that a Pascal Boolean is one byte long and a LOGICAL is four bytes long when the Boolean type stands alone, in a record structure, the Boolean must be one byte long.

To test the Boolean for TRUE and FALSE:

1. Use the ICHAR transfer function to convert the CHAR value to an integer.

2. Test for equivalence to 0. If the value is equivalent to 0, the Boolean value is FALSE.

See Section 1.7.5 for information about record structures.

1.7.2. Pointers

Throughout the documentation you will see references to a data type known as a pointer. A **pointer** is an address; it "points" to another data structure. A pointer is four bytes long. Many system calls return pointers as parameters. A common example is a call that returns a pointer to an array.

In the reference material a pointer may be described in one of three ways:

- With the phrase "in UNIV_PTR format".

- As being a pointer.

- As the address of a structure.

DOMAIN FORTRAN provides the POINTER statement as an extension to the ANSI standard, in order to make using returned pointers easier.

The POINTER statement permits you to access the data to which an address points. The syntax is:

POINTER /pointer-variable/based-variable-list

Where:

pointer-variable — Must be defined as an INTEGER*4 before you refer to it in the POINTER statement. (A pointer is a 32-bit address.)

based-variable-list — Lists variable(s) pointed to by the pointer-variable. If the pointer-variable points to a record structure, you specify all the variables that make up the record structure, in low byte to high byte order. (Section 1.7.5 describes how to emulate record structures.) The pointer refers directly to the variable listed first. Subsequent variables in the list are offset by the sum of the sizes of the previous variables, so that once the pointer variable is loaded, you may directly access any listed variable.

If the pointer points to an array, you may dimension the array in the POINTER statement.

PGM_$GET_ARGS is an example of a system call that returns a pointer. PGM_$GET_ARGS retrieves command line arguments. It places each argument in a record, preceded by the length of the argument. PGM_$GET_ARGS then loads an array with pointers to each record. PGM_$GET_ARGS returns two parameters, the number of arguments it has retrieved, and a pointer to the array of pointers.

Figure 1-1 illustrates the GET_ARG pointer arrangement:

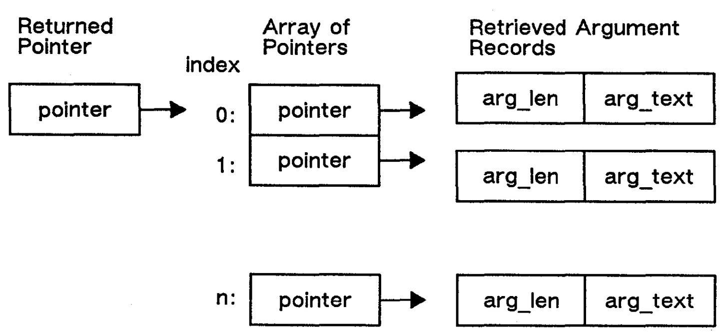


Figure 1-1. The Pointer/Data Relationship

The argument record structure consists of a 2-byte integer in the low end and a character string of up to 128 characters in the high end. The character string is the text of the argument, and the integer is the length of the argument.

The following program example uses the PGM_$GET_ARGS call to illustrate how to handle pointers in FORTRAN.

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pgm.ins.ftn'

      CHARACTER*128 arg_text
*     Declare pointers as 4-byte integers
      INTEGER*4 argv_ptr, { Pointer to array of args }
     2          arg_ptr,  { Pointer to record }
     2          argv
      INTEGER*2 arg_count,
     2          arg_len,
     2          i
*     Associate pointer and based list
      POINTER /argv_ptr/argv(0:127)      { Pointer to array }
      POINTER /arg_ptr/arg_len, arg_text  { Pointer to record }

*     Load argument records and pointer array
      CALL pgm_$get_args (arg_count,
     2                   argv_ptr)

*     Print out commmand line arguments
      DO 10 i = 0, arg_count-1
*     Associate ptr variable and ptrs in array
      arg_ptr = argv(i)
      write(*,*) 'argument ',i, ' is ', arg_text(1:arg_len)
10    CONTINUE

      END
```

Once a value has been assigned to a pointer, you can reference its based variables. In the example, the system assigns the address of the array to argv_ptr, which allows you to reference argv. You must explicitly assign each address in the array to the argument pointer, arg_ptr:

arg_ptr = argv(i)

This permits you to reference the variables in the argument record.

1.7.3. Enumerated Types

Pascal implements a data type known as an enumerated type, in which the type is associated with a list of values. A variable defined to be of this data type can only take one of these values.

In the reference material, the parameter description for an enumerated type ends with the sentence:

> Specify only one of the following predefined values: (for input parameters)
>
> or
>
> One of the following predefined values: (for output parameters)

This sentence is followed by a list of the predefined values that a variable of this type may hold. These values are defined by the subsystem insert file, and each corresponds to the ordinal position of the value in the data type definition.

To use an enumerated type in FORTRAN, define the parameter variable as a 2-byte integer, and load the variable using the predefined values listed in the parameter description.

The following is the description of the weekday parameter to the CAL_$WEEKDAY call:

weekday

The computed day of the week, in CAL_$WEEKDAY_T format.
This is a 2-byte integer. One of the following predefined values:

CAL_$SUN
CAL_$MON
CAL_$TUE
CAL_$WED
CAL_$THU
CAL_$FRI
CAL_$SAT

The following program example calls CAL_$WEEKDAY to determine what day of the week a specific date falls on. It uses the predefined values to determine what has been returned.

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/cal.ins.ftn'

      INTEGER*2 year,
     2          month,
     2          day,
     2          weekday

*     Get the input
      print *, 'What year? '
      read (*,10) year
      print *, 'What month? '
      read (*,10) month
      print *, 'What day? '
      read (*,10) day
10    format (BN,I3)
      weekday = cal_$weekday (year,
     2                        month,
     2                        day)

      IF (weekday .EQ. cal_$mon) THEN
         print *, 'The day of the week is Monday'
         ELSE IF (weekday .EQ. cal_$tue) THEN
         print *, 'The day of the week is Tuesday'
         ELSE IF (weekday .EQ. cal_$wed) THEN
         print *, 'The day of the week is Wednesday'
         ELSE IF (weekday .EQ. cal_$thu) THEN
         print *, 'The day of the week is Thursday'
         ELSE IF (weekday .EQ. cal_$fri) THEN
         print *, 'The day of the week is Friday'
         ELSE IF (weekday .EQ. cal_$sat) THEN
         print *, 'The day of the week is Saturday'
         ELSE IF (weekday .EQ. cal_$sun) THEN
         print *, 'The day of the week is Sunday'
      END IF
```

1.7.4. Sets

Another Pascal data type you must emulate is a set. A set is a bit field. In the reference material, the parameter description for a set ends with the sentence:

> Specify any combination of the following predefined values:

This sentence is followed by a list of predefined bit values to be used in setting the bit field. These values are defined by the subsystem insert file, and each corresponds to the position of a bit.

In FORTRAN, the bit field is always an integer variable. The parameter description will explicitly state whether it is a 2-byte or 4-byte integer.

There are some exceptions to this case. One is the MBX_$CHANNEL_SET_T data type, used to indicate channel numbers in a call to MBX_$GET_REC_CHAN_SET, and another is the GPR_$KEYSET_T data type, used to specify a set of keys in a call to GPR_$ENABLE_INPUT. These exceptions can be handled using set emulation calls supplied in the FTNLIB library. See Section 1.7.4.3 for information about the set emulation calls.

1.7.4.1. Setting Bits

In some cases you must set bits in a field that you pass to the system. The following is the description of the options parameter to the PGM_$ESTABLISH_FAULT_HANDLER call.

> options
>
> A value specifying the type of handler you want to establish, in PFM_$FH_OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following set of predefined values:
>
> PFM_$FH_MULTI_LEVEL
> To declare a multi-level fault handler which handles faults for its own program level and all subordinate levels.
>
> PFM_$FH_BACKSTOP
> To establish a backstop fault handler which takes effect after all non-backstop handlers have taken effect.

In this case, you declare the options parameter to be an INTEGER*2, and assign a value to it by adding the predefined values:

```
%include '/sys/ins/pfm.ins.ftn'

*     Declare the variable
      INTEGER*2 options

*     Set both bits
      options = pfm_$fh_multi_level + pfm_$fh_backstop

*     Use the parameter in a (function) call
      handle = pfm_$establish_fault_handler (t_status,
                                             options,
                                             func_p,
                                             status)
```

1.7.4.2. Testing Bits

In some cases the system returns a bit field that you must test to determine which bits are set. SIO_$INQUIRE returns an option parameter that may return the SIO_$ERR_ENABLE option. This option is a 2-byte bit field that may have the predefined values:

SIO_$CHECK_PARITY
SIO_$CHECK_FRAMING
SIO_$CHECK_DCD_CHANGE
SIO_$CHECK_CTS_CHANGE

To test a single bit (or test each bit separately):

1. AND the returned value and the predefined bit value.

2. If the result is 0, the bit is not set.

The following program segment calls SIO_$INQUIRE, asking which types of errors are enabled. SIO_$INQUIRE returns a bit field, which the program tests bit-by-bit to determine the types of errors that are enabled.

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/streams.ins.ftn'
%include '/sys/ins/sio.ins.ftn'

      INTEGER*4   status
      INTEGER*2   value_m,                 { Bit field }
                  stream_id

*     OPEN an SIO line w/ STREAM_$OPEN
                  .
                  .
                  .
*     INQUIRE enabled errors
      CALL sio_$inquire (stream_id,
     2                   sio_$err_enable,  { Option }
     2                   value_m,          { Specify bit mask }
     2                   status)

      IF (status .NE. status_$ok)
     2     GOTO ERROR
```

```
*      Test each bit and print enabled errors
       IF ( AND(value_m,sio_$check_parity) .NE. 0)
     2     print *, 'Parity errors enabled'

       IF ( AND(value_m,sio_$check_framing) .NE. 0)
     2     print *, 'Framing errors enabled'

       IF ( AND(value_m,sio_$check_dcd_change) .NE. 0)
     2     print *, 'DCD line changes reported'

       IF ( AND(value_m,sio_$check_cts_change) .NE. 0)
     2     print *, 'CTS line changes reported'
```

To test a number of specific bits:

1. Create a mask and set the bits you wish to test, using the predefined values.

2. AND the mask and the returned value. The AND results in a bit field in which the bits you set in the mask are either set or not, depending on the state of the corresponding returned value bits. That is, if bit 5 of the returned value was set, bit 5 in the result is set.

3. Test the bits using the predefined constants. If you want to test a bit for being set, add the predefined value to the value against which you test the result. If you want to test a bit for being not set, simply omit it from the test value.

The following program segment again calls SIO_$INQUIRE, asking which types of errors are enabled. In this case, it tests two bits for two specific conditions:

1. Both bits set.

2. One bit set, one bit not set.

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/streams.ins.ftn'
%include '/sys/ins/sio.ins.ftn'

       INTEGER*4    status
       INTEGER*2    mask
       INTEGER*2    value_m,                { Bit mask }
                    stream_id

*      OPEN an SIO line w/ STREAM_$OPEN
                    .
                    .
                    .
*      INQUIRE enabled errors
       CALL sio_$inquire (stream_id,
     2                    sio_$err_enable, { Option }
     2                    value_m,         { Specify bit mask }
     2                    status)

*      Create the mask
       mask = sio_$check_parity + sio_$check_framing
```

```
*     Test for both bits set

      IF ( AND(mask,value_m) .EQ.
     2     (sio_$check_parity + sio_$check_framing) )
     2    print *, 'Parity and Framing enabled'

*     Test for parity off, framing on

      IF ( AND(mask,value_m) .EQ. sio_$check_framing)
     2    print *, 'Parity not enabled - Framing enabled'
```

1.7.4.3. Emulating Large Sets

Two cases exist for which the set emulation techniques described above will not work; the MBX_$CHANNEL_SET_T data type (used to indicate channel numbers in a call to MBX_$GET_REC_CHAN_SET), and the GPR_$KEYSET_T data type (used to specify a set of keys in a call to GPR_$ENABLE_INPUT).

In both cases, there are no predefined values for the bits. MBX_$CHANNEL_SET_T is a set of integers from 0 to 255. GPR_$ENABLE_INPUT is a set of characters not exceeding 256.

To initialize, set, clear, and test these sets, use the set emulation calls supplied in the FTNLIB library.

To initialize a set, use the LIB_$INIT_SET call with the following syntax:

LIB_$INIT_SET(name-of-set, number-of-elements-in-set)

A set should be initialized before using it.

To set a bit in a set, use the LIB_$ADD_TO_SET call with the following syntax:

LIB_$ADD_TO_SET(name-of-set,number-of-elements-in-set,new-element)

LIB_$ADD_TO_SET must be called once for each element you wish to add to the set.

To clear a bit from a set, use LIB_$CLR_FROM_SET call with the following syntax:

LIB_$CLR_FROM_SET(name-of-set,number-of-elements-in-set,element-to-clear)

LIB_$CLR_FROM_SET must be called once for each element you want to clear from the set.

To test a bit in a set, use the LIB_$MEMBER_OF_SET call with the following syntax:

boolean = LIB_$MEMBER_OF_SET(name-of-set,number-of-elements-in-set,
element-to-test)

The Boolean value returns TRUE if the tested element is in the set.
The following program example declares the channel set as an 8-element INTEGER*4 array. This creates a bit field of 255 bits -- each bit corresponds to a channel number. The program uses the set emulation calls to specify that messages be accepted from two channels - 2 and 4.

```
%INCLUDE '/sys/ins/base.ins.ftn'
%INCLUDE '/sys/ins/mbx.ins.ftn'

      INTEGER*4 handle, status, retptr, retlen
      INTEGER*2 buffer(4), returned_buffer(4),open_channels
      POINTER /retptr/returned_buffer
      INTEGER*4 chanset(8)  { Declare channel # set (265 bits) }

      { Initialize the set. }
      CALL lib_$init_set(chanset,      { Set name }
     2                   int2(256) ) { Number of elements }

      { Set channel 2. }
      CALL lib_$add_to_set(chanset,   { Set name }
     2                     int2(256), { Number of elements }
     2                     int2(2) )  { Element to set -- channel 2}

      { Set channel 4. }
      CALL lib_$add_to_set(chanset,
     2                     int2(256),
     2                     int2(4) )  { Element to set -- channel 4}

*    Create the mailbox -- ten communication channels, 100 bytes in
*    the queue.

      open_channels = 0
      CALL mbx_$create_server('mailbox',
     2                         int2(7),
     2                         int2(100),
     2                         int2(10),
     2                         handle,
     2                         status)
      CALL error('mbx_$create_server',status)
      write(*,*) 'Mailbox opened.'

*    Get the messages

100   CALL mbx_$get_rec_chan_set(handle,
     2                           chanset,      { Channel set }
     2                           iaddr(buffer),
     2                           int4(8),
     2                           retptr,
     2                           retlen,
     2                           status)
                 .
                 .
                 .
```

1.7.5. Records

A **record** is a complex data structure encoded into a single variable. A record may be composed of several "fields" of information that can be referenced separately. Records are of differing sizes, depending on the information being transferred.
The reference material is useful in determining how to emulate records.

The parameter description for a record will end with the sentence: "This data type is X bytes long. See the XXX Data Types for more information." This sentence tells you the length of the record, in bytes, and references you to the appropriate subsystem Data Types section.

As described in Section 1.6.1, each record is illustrated in the Data Types section, in order to make it easier for you to understand what it is you wish to emulate.

The illustrations show the size and type of each field, and describe the fields that make up the record. The following is the illustration of the CAL_$TIMEDATE_REC_T predefined record:

CAL_$TIMEDATE_REC_T	Readable time format. The diagram below illustrates the CAL_$TIMEDATE_REC_T data type:

predefined type	byte: offset		field name
	0:	integer	year
	2:	integer	month
	4:	integer	day
	6:	integer	hour
	8:	integer	minute
	10:	integer	second

This record may be passed to the system using the CAL_$ENCODE_TIME call, or returned from the system using the CAL_$DECODE_TIME call.

Typically, you use an array to emulate a record, and you use EQUIVALENCE statements to access the record's fields as separate variables.

The following program segment accepts the six fields of the CAL_$TIMEDATE_REC_T record as separate input variables, and passes the full record to CAL_$ENCODE_TIME as a 6-element 2-byte integer array. It does so by equivalencing each field to an element of the array.

```
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/time.ins.ftn'
%include '/sys/ins/cal.ins.ftn'

*     Emulate cal_$timedate_rec_t
      INTEGER*2 c_clock(6),        { Array -- full record }
     2          year,              { Six separate fields }
     2          month,
     2          day,
     2          hour,
     2          minute,
     2          second

*     Equivalence each element with a field
      EQUIVALENCE (c_clock(1),year),
     2            (c_clock(2),month),
     2            (c_clock(3),day),
     2            (c_clock(4),hour),
     2            (c_clock(5),minute),
     2            (c_clock(6),second)


*     Emulate time_$clock_t
      INTEGER*2 clock(3)

*     Get input variables
      WRITE (*,*) 'Input year in integer format: '
      READ  (*,10) year
                .
                .
                .
      WRITE (*,*) 'Input second in integer format: '
      READ  (*,60) second
10    FORMAT (BN,I3)

*     Convert  TIMEDATE_REC_T to CLOCK_T
      CALL cal_$encode_time (c_clock,
     2                       clock)
```

1.7.6. Variant Records

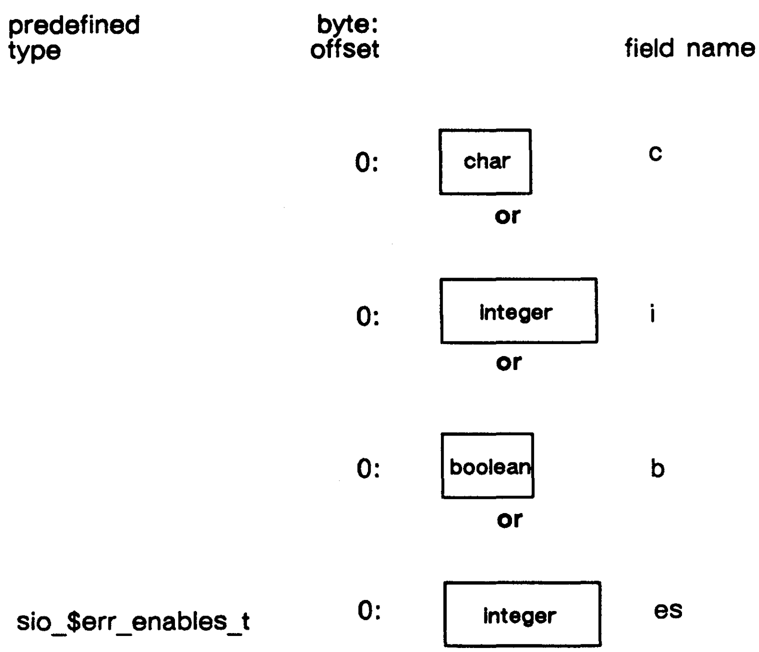
Pascal implements a special type of record, the variant record, in which the definition of the record may differ, depending on the value of a field in the record or the record's usage. An example of this is the SIO_$VALUE_T predefined type.

This record may alternately be a character, a positive 2-byte integer, a Boolean value, or a set (bit field).

In the Data Types section of the reference material, all possible variations are illustrated.

One way to emulate a variant type is to declare the parameter to be whichever form you wish to reference. In cases where you wish to reference the parameter in more than one form, declare more than one variable and use each form where appropriate.

Below is the data type description of the variant record SIO_$VALUE_T.



Field Descriptions

c
A character value.

i
An integer value.

b
A boolean value.

es
A set of enabled errors.

The following program segment uses the SIO_$INQUIRE call to determine several options for a serial line. The value returned by this call is in the format SIO_$VALUE_T and may be a 2-byte integer, a Boolean value, a character value, or a bit field, depending on which option is being inquired. The program declares variables of all four types and uses whichever is appropriate to the specific call.

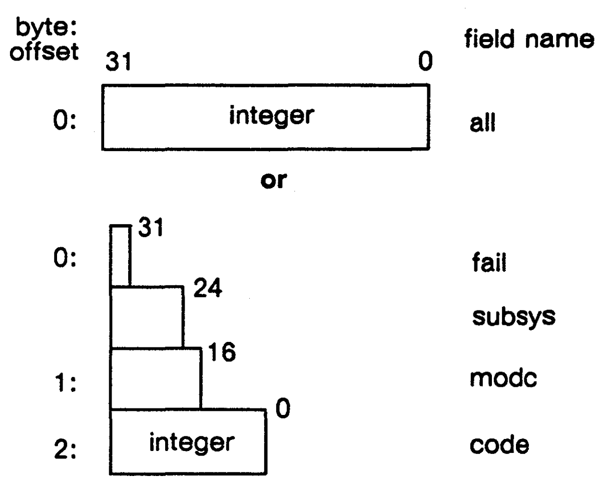
```
*  This program inquires and changes attributes of a serial line

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/streams.ins.ftn'
%include '/sys/ins/sio.ins.ftn'
%include '/sys/ins/error.ins.ftn'

*      $OPEN variables
       INTEGER*4      status
       CHARACTER*256  pathname
       INTEGER*2      namelength,
     2                stream_id,
     2                access,
     2                conc
```

```
*     Declare 4 forms of the parameter
      LOGICAL      value_b    { Boolean value }
      CHARACTER*1 value_c    { Character value }
      INTEGER*2    value_i,   { Integer value }
     2             value_m    { Bit field }

*     Get pathname as input
      print *,'Input the pathname'
      read (*,10) pathname
10    format (BN,A80)
      namelength = LEN(pathname)

      CALL stream_$open (pathname,
     2                   namelength,
     2                   stream_$write,         { Access }
     2                   stream_$no_conc_write, { Concurrency }
     2                   stream_id,
     2                   status)

      IF (status .NE. status_$ok)
     2    GOTO ERROR

*     INQUIRE serial line # (INTEGER)
      CALL sio_$inquire (stream_id,
     2                   sio_$line,             { Option }
     2                   value_i,
     2                   status)

      IF (status .NE. status_$ok)
     2    GOTO ERROR

      write (*,30) value_i
30    format ('The serial line is ',I3)


*     INQUIRE if CTS is enabled (BOOLEAN)
      CALL sio_$inquire (stream_id,
     2                   sio_$cts_enable,       { Option }
     2                   value_b,
     2                   status)

      IF (status .NE. status_$ok)
     2    GOTO ERROR

      write (*,40) value_b
40    format ('The CTS_ENABLE is ',L5)

*     INQUIRE the KILL char (CHARACTER)
      CALL sio_$inquire (stream_id,
     2                   sio_$kilL,             { Option }
     2                   value_c,
     2                   status)

      IF (status .NE. status_$ok)
     2    GOTO ERROR
```

```
*      Test for ^X using  hex value
       IF (ICHAR(value_c) .EQ. 16#18) THEN
           print *, 'The KILL character is control X'
       ELSE
           GOTO ERROR
       ENDIF

*      INQUIRE which errors are enabled (MASK)
       CALL sio_$inquire (stream_id,
      2                   sio_$err_enable,        { Option }
      2                   value_m,
      2                   status)

       IF (status .NE. status_$ok)
      2    GOTO ERROR

*      Test each bit and print if set
       IF ( AND(value_m, sio_$check_parity) .NE. 0)
      2    print *, 'Parity errors enabled'

       IF ( AND(value_m, sio_$check_framing) .NE. 0)
      2    print *, 'Framing errors enabled'

       IF ( AND(value_m, sio_$check_dcd_change) .NE. 0)
      2    print *, 'DCD line changes reported'

       IF ( AND(value_m, sio_$check_cts_change) .NE. 0)
      2    print *, 'CTS line changes reported'
```

You may also equivalence the variants. The status returned from system calls is a variant type. Typically, after each call you test the status.all form (the full four bytes) against the success status, STATUS_$OK. However, when checking for a STREAM_$END_OF_FILE status, you test against the status.code form of the record.

Below is the data type description of the STATUS_$T type.



all
All 32 bits in the status code.

code
A signed number that identifies the
type of error that occurred (bits 0 - 15).

modc
The module that encountered the
error (bits 16 - 23).

subsys
The subsystem that encountered the
error (bits 24 - 30).

fail
The fail bit. If this bit is set, the error
was not within the scope of the module invoked,
but occurred within a lower-level module (bit 31).

The program segment below equivalences both variants and accesses whichever form of the status it needs.

```
*     Declare status
      INTEGER*2 status(2)
      INTEGER*4 status_all
      INTEGER*2 status_code

*     Declare GET_REC variables
            .
            .
            .
*     Open a file
            .
            .
            .
*     Read a record
      Call  STREAM_$GET_REC ( stream_id,
                              IADDR(info_rec), { Address of buffer }
                              LEN(info_rec),   { Length of buffer }
                              retptr,          { Pointer to returned data }
                              retlen,          { Length of returned }
                              seek_key,        { Returned seek key }
                              status);         { Returned status }

*     Test the returned status
      IF (status_all .NE. status_$ok) THEN
         IF (status_code .EQ. stream_$end_of_file) THEN
            GOTO CLOSE
         ENDIF
         GOTO ERROR
      ENDIF
```

1.7.7. Passing Parameters to System Calls

DOMAIN requires that integer variables and integer constants be of a particular length, depending on the usage of the parameter.

1.7.7.1. Passing Integer Parameters

When passing integer parameters to system calls, it is important to pass an integer that is the size that the call expects.

In the reference material, the second sentence of a parameter description informs you whether the expected integer is a 2-byte or 4-byte integer.

If you declare all your integer data types as INTEGER*4, it is important to note that some call parameters expect a 2-byte integer value; for example, pathname lengths.

To pass an integer to a system call that expects a 2-byte integer, either explicitly declare the parameter variable to be INTEGER*2, or typecast the parameter to be INTEGER*2 with the INT2 intrinsic function.

The two following examples show both ways of passing an integer properly. The NAME_$SET_DIR call permits you to set a naming directory by passing the pathname of the directory and the length of the pathname. The length parameter is expected to be a 2-byte integer. Example A declares the length parameter as a 2-byte integer. Example B declares the length parameter as a 4-byte integer, and typecasts the parameter in the call.

EXAMPLE A

```
      INTEGER*4     status
      CHARACTER*256 pathname
      INTEGER*2     namelength
                .
                .
                .
      CALL name_$set_ndir (pathname,
     2                     namelength,
     2                     status)
```

EXAMPLE B

```
      INTEGER*4     status
      CHARACTER*256 pathname
      INTEGER*4     namelength
                .
                .
                .
      CALL name_$set_ndir (pathname,
     2                     INT2(namelength),
     2                     status)
```

1.7.7.2. Passing Integer Constants

DOMAIN system calls permit you to specify integer constants as parameters where applicable. Again, it is important that when you do so, you are careful to pass a constant of the expected length.

In FORTRAN, integer constants have the same length as the default integer type (INTEGER*4). To pass a constant to a call that expects a 2-byte integer value, type cast the constant with the intrinsic function INT2.

1.8. Data Type Information for C Programmers

As stated above, DOMAIN predefined data types reflect the data types available in Pascal. However, you can use standard C programming statements to emulate the data types that are not supported.

In addition, the way that parameters are passed also reflects Pascal. That is, parameters are passed by reference rather than by value. In the C insert files, each system call is declared using the "std_$call" keyword that informs the compiler that your program will pass parameters to system calls by reference. Obviously, this will effect the way you specify parameters. Section 1.8.5 describes how to avoid problems when passing parameters to system calls.

The following four sections describe data types to be emulated. Each section explains:

- The purpose of the data type.
- How to recognize the type in the reference material.
- How to emulate the type using C.
- How to reference a variable of this type.

1.8.1. Boolean Type

Boolean types are variables that evaluate to either TRUE or FALSE. A Boolean value is described in the reference material and the insert files as a Boolean.

The C BASE insert file predeclares a *boolean* type, to emulate a Pascal Boolean type. It also declares a true and false value for use with the boolean type.

1.8.2. Sets

Another Pascal data type you must emulate is a set. A set is a bit field.

In the reference material, the parameter description for a set ends with the sentence:

> Specify any combination of the following predefined values:

This sentence is followed by a list of predefined bit values to be used in setting the bit field. These values are defined by the subsystem insert file, and each corresponds to the position of a bit.

In C, the bit field is usually an integer variable. However, the insert files predefine the bit field types so that you may use the predefined types listed in the parameter descriptions.

There are some exceptions to this case. One is the MBX_$CHANNEL_SET_T data type, used to indicate channel numbers in a call to MBX_$GET_REC_CHAN_SET, and another is the GPR_$KEYSET_T data type, used to specify a set of keys in a call to GPR_$ENABLE_INPUT. These exceptions can be handled using set emulation calls supplied in the FTNLIB library. See Section 1.8.2.3 for information about the set emulation calls.

1.8.2.1. Setting Bits

In some cases, you must set bits in a field that you pass to the system. The following is the description of the options parameter to the PGM_$ESTABLISH_FAULT_HANDLER call.

options

A value specifying the type of handler you want to establish, in PFM_$FH_OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following set of predefined values:

PFM_$FH_MULTI_LEVEL
To declare a multi-level fault handler which handles faults for its own program level and all subordinate levels.

PFM_$FH_BACKSTOP
To establish a backstop fault handler which takes effect after all non-backstop handlers have taken effect.

In this case, you declare the options parameter using the predefined type PFM_$FH_OPT_SET_T, and assign a value to it by adding the predefined bit values:

```
#include <stdio.h>
#include "/sys/ins/pfm.ins.c"

      /* Declare the variable. */
      pfm_$fh_opt_set_t options;
                .
                .
                .
      /* Set both bits. */
      options = pfm_$fh_multi_level + pfm_$fh_backstop;

      /* Use the parameter in a (function) call. */
      handle = pfm_$establish_fault_handler (t_status,
                                             options,
                                             func_p,
                                             status);
```

1.8.2.2. Testing Bits

In some cases, the system returns a bit field that you must test to determine which bits are set. SIO_$INQUIRE returns an option parameter that may return the SIO_$ERR_ENABLE option. This option is a 2-byte bit field that may have the predefined values:

SIO_$CHECK_PARITY
SIO_$CHECK_FRAMING
SIO_$CHECK_DCD_CHANGE
SIO_$CHECK_CTS_CHANGE

To test a single bit (or test each bit separately):

1. AND the returned value and the predefined bit value.

2. If the result is 0, the bit is not set.

The following program segment calls SIO_$INQUIRE, asking which types of errors are enabled. SIO_$INQUIRE returns a bit field (value.es), which the program tests bit-by-bit to determine the types of errors that are enabled.

```
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/sio.ins.c"
#include "/sys/ins/error.ins.c"

    status_$t       status;

    /* SIO_$ variables */
    sio_$value_t  value;
    stream_$id    stream_id;

    /* Open an SIO line with STREAM_OPEN. */
                  .
                  .
                  .

    /* INQUIRE enabled errors. */
    sio_$inquire (stream_id,
                  sio_$err_enable,                /* Option */
                  value.es,
                  status);

    if (status.all != status_$ok)
        error_$print (status);

    if ((value.es & sio_$check_parity) != 0)      /* Bit set */
       printf ("Parity errors enabled \n");

    if ((value.es & sio_$check_framing) != 0)     /* Bit set */
       printf ("Framing errors enabled \n");

    if ((value.es & sio_$check_dcd_change) != 0) /* Bit set */
       printf ("DCD line changes reported \n");
```

```
if ((value.es & sio_$check_cts_change) != 0) /* Bit set */
   printf ("CTS line changes reported \n");
```

To test a number of specific bits:

1. Create a mask and set the bits you wish to test, using the predefined values.

2. AND the mask and the returned value. The AND results in a bit field in which the bits you set in the mask are either set or not, depending on the state of the corresponding returned value bits. That is, if bit 5 of the returned value was set, bit 5 in the result is set.

3. Test the bits using the predefined constants. If you want to test a bit for being set, add the predefined value to the value against which you test the result. If you want to test a bit for being not set, simply omit it from the test value.

The following program segment again calls SIO_$INQUIRE, asking which types of errors are enabled. In this case, it tests two bits for two specific conditions:

1. Both bits set.

2. One bit set, one bit not set.

```
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/sio.ins.c"
#include "/sys/ins/error.ins.c"

    status_$t       status;

    /* SIO_$ variables */
    sio_$value_t  value;
    stream_$id    stream_id;

    /* Open an SIO line with STREAM_OPEN. */
              .
              .
              .

    /* INQUIRE enabled errors. */
    sio_$inquire (stream_id,
                  sio_$err_enable,       /* Option */
                  value.es,
                  status);

    if (status.all != status_$ok)
        error_$print (status);

    /* Create a mask. */
    mask = sio_$check_parity + sio_$check_framing;

    /* Test for both bits set. */
    if ((value.es & mask) ==
        ( sio_$check_parity + sio_$check_framing))
    printf ("Parity and Framing enabled \n");
```

```
/* Test for parity off, framing on. */
if ((value.es & mask) == sio_$check_framing);
printf ("Parity enabled - Framing not enabled \n");
```

1.8.2.3. Emulating Large Sets

Two cases exist for which the set emulation techniques described above will not work; the MBX_$CHANNEL_SET_T data type (used to indicate channel numbers in a call to MBX_$GET_REC_CHAN_SET), and the GPR_$KEYSET_T data type (used to specify a set of keys in a call to GPR_$ENABLE_INPUT).

In both cases, there are no predefined values for the bits. MBX_$CHANNEL_SET_T is a set of integers from 0 to 255. GPR_$ENABLE_INPUT is a set of characters not exceeding 256.

To initialize, set, clear, and test these sets, use the set emulation calls supplied in the FTNLIB library.

To initialize a set, use the LIB_$INIT_SET call with the following syntax:

    LIB_$INIT_SET(name-of-set, number-of-elements-in-set)

A set should be initialized before using it.

To set a bit in a set, use the LIB_$ADD_TO_SET call with the following syntax:

    LIB_$ADD_TO_SET(name-of-set,number-of-elements-in-set,new-element)

LIB_$ADD_TO_SET must be called once for each element you wish to add to the set.

To clear a bit from a set, use LIB_$CLR_FROM_SET call with the following syntax:

    LIB_$CLR_FROM_SET(name-of-set,number-of-elements-in-set,element-to-clear)

LIB_$CLR_FROM_SET must be called once for each element you wish to clear from the set.

To test a bit in a set, use the LIB_$MEMBER_OF_SET call with the following syntax:

    boolean = LIB_$MEMBER_OF_SET(name-of-set,number-of-elements-in-set,
                                  element-to-test)

The Boolean value returns TRUE if the tested element is in the set.

The following program example declares the channel set in the usual way, using the predefined MBX_$CHANNEL_SET_T type. This creates a bit field of 255 bits - each bit corresponds to a channel number. The program uses the set emulation calls to specify that messages be accepted from two channels -- 2 and 4.

```
#include </sys/ins/base.ins.c>
#include </sys/ins/mbx.ins.c>
#include </sys/ins/error.ins.c>
        .
        .
        .
/* Declare channel set. */
mbx_$chan_set_t     chan_set;

main()  /* Program server */
{
    init();

/* Create the mailbox. */
    mbx_$create_server( mbx_name,
                        mbx_namelen,
                        mbx_chansize,
                        mbx_maxchan,
                        mbx_handle,
                        status );
    if(status.all != 0)
    {   error_$print_name( status, "error creating mailbox" , 22 );
        exit(1);
    }
    printf("Mailbox %s was successfully opened.\n",mbx_name);

/* Initialize set. */
    lib_$init_set(chan_set,         /* Name */
                  256);             /* Number of elements */

/* Set channel 2. */
    lib_$add_to_set(chan_set,       /* Name */
                    256,            /* Number of elements */
                    2);             /* Channel # to set */

    lib_$add_to_set(chan_set,
                    256,
                    4);

/* Keep getting messages until there are no more clients. */
    do
    {   mbx_$get_rec_chan_set(
                      mbx_handle,
                      chan_set,     /* Channel set */
                     &srv_msg_buf,
                      srv_msg_len,
                      mbx_retptr,
                      mbx_retlen,
                      status );
        if (status.all != 0)
        {   error_$print_name( status, "error getting record" , 20 );
            return(1);
        }
        printf("Message received from channel %4d\n",mbx_retptr->mbx_hdr.chan);
          .
          .
          .
```

1.8.3. Records

A Pascal record is analogous to a C structure. Both may be composed of several "fields" of information that can be referenced separately.

The C insert files predefine structures to emulate the records required by system calls.

In the reference material, if a parameter has a predefined record type, the first sentence of the description ends with the phrase, "in XXX format", where XXX is the predefined type.

For example, the CAL_$DECODE_LOCAL_TIME system call has one parameter, decoded_clock. The following is the parameter description:

```
decoded_clock

  The local time, in CAL_$TIMEDATE_REC_T format.  This is a 6-element
  array of 2-byte integers. The first element represents the year,
  the second the month, and so on.
```

The following program segment declares and loads this record, then accesses one field in it:

```
#include "/sys/ins/base.ins.c";
#include "/sys/ins/cal.ins.c";

    cal_$timedate_rec_t d_clock;
              .
              .
              .
    /* Get decoded local time -- load d_clock. */
    cal_$decode_local_time (d_clock);

    /* Access the year. */
    printf ("The year is %s \n",d_clock.year);
```

To determine the field names of predefined records, see the illustrations in the appropriate Data Types section, or read the appropriate insert file.

1.8.4. Variant Records

A Pascal variant record permits a single field of a record to contain any one of several data types, depending on usage. A Pascal variant record can be emulated by using C unions.

The C insert files predefine structures to emulate the variant records required by system calls. In the reference material, if a parameter has a predefined variant record type, the first sentence of the description ends with the phrase, "in XXX format", where XXX is the predefined type.

For example, the status parameter returned by most system calls is a variant record, in STATUS_$T format. The following program declares status parameter, loads it by calling the system call, then accesses it in two different forms.

```
#include "/sys/ins/base.ins.c";
#include "/sys/ins/error.ins.c";

   name_$pname_t name;
   short         len;
   status_$t     status;
         .
         .
         .
   name_$get_ndir(name,
                  length,
                  status);

   /* Check status. */
   if (status.all != status_$ok)                      /* Access one form */
       printf(" status code is: %d", status.code)     /* Access another form */
```

To determine the field names of predefined records, see the illustrations in the appropriate Data Types section, or read the appropriate insert file.

NOTE: DOMAIN C permits you to reference members of structures or unions that are inside other structures or unions without specifying all of the member names.

1.8.5. Passing Parameters to System Calls

As discussed above, parameters are passed to DOMAIN system calls by reference. Because of this, you must pay particular attention to the way you declare and pass character arrays.

In addition, DOMAIN requires that integer variables and integer constants be of a particular length, depending on the usage of the parameter.

NOTE: If a call has no parameters, you must specify an empty set of parentheses for the call to work properly.

1.8.5.1. Passing Character Arrays

The way that you pass a character array to a system routine depends on how the array was declared. In C, a character array may be declared two ways:

1. As a "true" array, using the following syntax:

   char example_array[25];

2. As a pointer to a character array, using the following syntax:

   char *example_array;

In the insert files, all character arrays are declared as "real" arrays. For example, the following definition of the NAME_$PNAME_T data type appears in the BASE insert file:

```
#define name_$pnamlen_max 256    /* Max pathname length */
typedef char name_$pname_t[name_$pnamlen_max];
```

If you declare a pathname using the predefined type, specify the parameter as follows:

```
status_$t      status;
short          len;
name_$pname_t  pathname;     /* Declared using predefined type */
               .
               .
               .
name_$set_ndir (pathname,    /* Passed by reference */
                len,
                status);
```

If you declare a pathname using the pointer syntax, you must dereference the pointer before you pass it. Specify the parameter as follows:

```
status_$t      status;
short          len;
char           *pathname;    /* Pointer syntax */
               .
               .
               .
name_$set_ndir (*pathname,   /* De-reference the pointer */
                len,
                status);
```

Because the system call is a "std_$call", it expects the parameter to be passed by reference. If you do not dereference the pointer before you pass it, an extra (incorrect) level of indirection is introduced.

NOTE: When the system returns a character array, it may not be null-terminated. If you intend to use it as a string, you must explicitly null-terminate it or use the length that the system returns as well.

1.8.5.2. Passing Integer Parameters

When passing integer parameters to system calls, it is important to pass an integer that is the size that the call expects.

In the reference material, the second sentence of a parameter description informs you whether the expected integer is a 2-byte or 4-byte integer.

If you declare all your integer data types as "int", it is important to note that an "int" type on the DOMAIN system is a 32-bit integer -- *not* a 16-bit integer.

To pass an integer to a system call that expects a 2-byte integer, either explicitly declare the parameter variable to be a "short" type, or type cast the "int" parameter to be short. The two following examples show both ways of passing an integer properly. The NAME_$SET_DIR call permits you to set a naming directory by passing the pathname of the directory and the length of the pathname. The length parameter is expected to be a 2-byte integer. Example A declares the length parameter as a 2-byte integer. Example B declares the length parameter as a 4-byte integer, and typecasts the parameter in the call.

EXAMPLE A

```
status_$t      status;
short          len;          /* Declared to expected size */
name_$pname_t  pathname;

name_$set_ndir (pathname,
                len,
                status);
```

EXAMPLE B

```
status_$t      status;
int            len;
name_$pname_t  pathname;

name_$set_ndir (pathname,
                (short)len,  /* Type cast to expected size */
                status);
```

There is a third case to consider. If you use the "strlen" function to load the length of a character array, note that it always returns a 4-byte integer. Again, you must either type cast this returned value or declare the returned value as a short integer and force strlen to load the 4-byte value into a 2-byte variable. Example A typecasts the value that strlen returns as the length of the pathname. Example B forces strlen to load the returned value in a short integer.

EXAMPLE A

```
status_$t      status;
name_$pname_t  pathname;

name_$set_ndir (pathname,
                (short)strlen(pathname),
                status);
```

EXAMPLE B

```
status_$t      status;
short          len;         /* Declared to expected size */
name_$pname_t  pathname;

/* Force strlen to return into 2 bytes. */
len = strlen(pathname)

name_$set_ndir (pathname,
                len,
                status);
```

1.8.5.3. Passing Integer Constants

DOMAIN system calls permit you to specify integer constants as parameters, where applicable. Again, it is important that when you do so, you are careful to pass a constant of the expected length.

Normally, the C compiler considers all constants as 4-byte entities. However, in DOMAIN system calls, any constant between the values -32768 and 32767 is passed as a 2-byte entity. This is done because DOMAIN system calls most commonly expect 2-byte values where constants can be used (i.e, the length of names).

If you are passing a constant to a call that expects a 4-byte integer value, you must type cast the constant to be long. Use a long constant (i.e., 20L) to typecast a constant to be long.

Chapter 2
How to Handle Errors and Faults

Any serious programming effort should include a method of handling runtime errors. Runtime errors take two forms:

System errors — Error condition returned from system calls and detected by the algorithms of your program. For example, passing an invalid parameter to a system call results in a system error.

Faults — Error condition detected (usually) by the hardware. For example, an attempt to access protected memory results in an access violation fault.

The first half of this chapter describes how to detect system errors, and how to format and print the corresponding error messages, using the ERROR system calls. The second half of the chapter describes how to handle faults, using the PFM system calls.

2.1. System Calls, Insert Files, and Data Types

To format and print errors, use system calls with the prefix ERROR. In order to use ERROR system calls, you must include the appropriate insert file in your program. The ERROR insert files are:

/SYS/INS/ERROR.INS.C	for C programs.
/SYS/INS/ERROR.INS.FTN	for FORTRAN programs.
/SYS/INS/ERROR.INS.PAS	for Pascal programs.

To handle faults, use the system calls with the prefix PFM. You must also include the appropriate insert file. The PFM insert files are:

/SYS/INS/PFM.INS.C	for C programs.
/SYS/INS/PFM.INS.FTN	for FORTRAN programs.
/SYS/INS/PFM.INS.PAS	for Pascal programs.

This chapter is intended to be a guide for performing certain programming tasks; the data type and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference*.

2.2. Status Structure

Most DOMAIN system calls return a 32-bit integer status code. A **status code** indicates the condition in which the call completed. If a call succeeds, the value of the status code is 0. If the call fails, the returned status will vary, depending on the nature of the failure.

The structure of a status code permits it to convey several pieces of information. A status code is a variant record, in STATUS_$T format. Figure 2-1 shows a diagram of this data type:

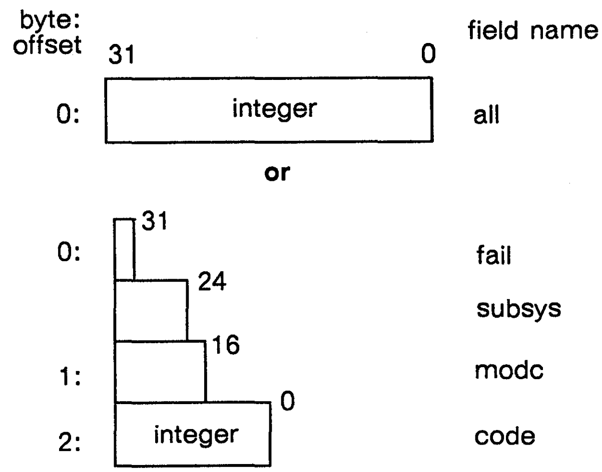


Figure 2-1. The Structure of the Status Data Type

If a call fails, each of the fields contains the following:

all	The full status - usually used to test for successful completion.
fail	The fail bit -- if this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.
subsys	The subsystem code -- a number identifying the subsystem that encountered the error.
modc	The module code -- a number identifying the module that encountered the error. (Some subsystems, such as STREAMS, are made up of several modules.)
code	The error code -- a signed number identifying the type of error that occurred. Each type of error is associated with a unique number.

The subsystem code, module code, and error code are all associated with text strings. The text associated with the error code explains the nature of the error, while the text associated with the module and subsystem are the names of each. You use a number of the DOMAIN ERROR subsystem calls to access these text strings.

2.2.1. Accessing Fields of the Status Code with FORTRAN

Four ERROR routines exist specifically for FORTRAN users to access each of the fields that make up a status code. They are:

```
code = ERROR_$CODE (status)
fail = ERROR_$FAIL (status)
module = ERROR_$MODULE (status)
subsys = ERROR_$SUBSYS (status)
```

CODE, MODULE, and SUBSYS take a status code as an input parameter and return the respective piece of the code as a 2-byte integer. FAIL takes a status code as an input parameter and returns a LOGICAL value indicating whether the fail bit is set.

2.3. Testing for Errors

When a system call returns a status to your program, you should always examine the returned status. In general, when testing a status code you should test the full 32-bit code. For Pascal and C users, that is status.all; FORTRAN users should declare status to be an INTEGER*4 variable and test the full value.

The insert file for a subsystem declares a mnemonic constant for each of the status codes that the subsystem may return. For example, the BASE insert file declares the constant STATUS_$OK to be equivalent to the success status 0. Always use the mnemonic constants when referencing status codes.

Typically, you test the returned status for success and, if the call failed, print an explanatory error message before exiting. Below is a program segment that tests the STREAM_$DELETE call for the success status:

```
STREAM_$DELETE (stream-id,
                status);

{ Test the returned staus. }
IF status.all <> status_$ok THEN
{ Print an error message. }
```

Printing error messages is described in the next section.

2.4. Printing Error Messages

The simplest way of printing an error message is to use ERROR_$PRINT. This call takes the status as input and prints out the text associated with the error code, along with the subsystem and module names.

Example 2-1 demonstrates a simple error-handling procedure. (It is the error-handling procedure invoked in many of the examples in this book.) Note that the procedure uses PGM_$EXIT to exit. PGM_$EXIT will exit from within a subroutine (if necessary), close any open files, release any acquired storage, and call PGM_$SIGNAL (to invoke any clean-up handlers) before exiting.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
    status    : STATUS_$T;
    { Declare CREATE variables }
          .
          .
          .
```

Example 2-1. A Simple Error-Handling Procedure

```
{ Declare procedure for error_handling. }
PROCEDURE error_routine;

    BEGIN
      error_$print (status);
      pgm_$exit;
    END;  { error_routine }

BEGIN { Main Program }

   { Create a file. }
   stream_$create (pathname,
                   namelength,
                   access,
                   conc,
                   stream_id,
                   status);

   { Test the returned staus. }
   IF status.all <> status_$ok THEN
      { Invoke error handling procedure.}
      error_routine;
          .
          .
          .
```

Example 2-1. A Simple Error-Handling Procedure (Cont.)

This program produces the following error message format:

file already exists (stream_$write specified on create) (stream manager/open)

The last section of the error indicates that the error status was passed from the open module of the stream manager.

2.5. Standardized Error Reporting

DOMAIN-supplied software follows these standards for error reporting:

- Reports all errors on STREAM_$ERROUT.
- Uses a question mark as a prefix character.
- Prints any filenames in lowercase surrounded by double quotation marks.

For example, the following is an error returned from the CPF Shell command.

?(cpf) "file.dat" - name not found (OS/naming server)

By using the system calls ERROR_$INIT_STD_FORMAT, ERROR_$STD_FORMAT, and ERROR_$PRINT_FORMAT, you may standardize the format of your error reporting along the same lines. These routines permit you to specify:

- The stream on which to report errors.
- A prefix character.
- A program name to appear in parentheses.
- Text of the error message.

ERROR_$PRINT_FORMAT permits you to specify all of the above with one system call. ERROR_$INIT_STD_FORMAT and ERROR_$STD_FORMAT work in conjunction with each other to specify the same type of error message. Calling ERROR_$INIT_STD_FORMAT and ERROR_$STD_FORMAT is equivalent to calling ERROR_$PRINT_FORMAT. However, for programs that use common subroutines, the former method provides more flexibility. For example, if an application's command level sets the command name with ERROR_$INIT_STD_FORMAT, it automatically provides the common lower-level modules with the correct command name for their error messages. Also, because ERROR_$STD_FORMAT has fewer parameters, it is easier to code using the pair of calls instead of using ERROR_$PRINT_FORMAT. ERROR_$STD_FORMAT uses a VFMT-style control string (see Chapter 8 for information about how to construct a control string).

The program in Example 2-2 uses ERROR_$INIT_STD_FORMAT and ERROR_$STD_FORMAT to print an error message that simulates standard error format. The program prints the error message in the main program to avoid passing parameters to the error procedure.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
    status : status_$t;

    { Declare CREATE variables. }
            .
            .
            .
    { Declare procedure for error_handling. }
PROCEDURE error_routine;
   BEGIN
      pgm_$set_severity (pgm_$error);
      pgm_$exit;
   END;  { error_routine }

BEGIN  { Main Program }
            .
            .
            .
   { Initialize standard error format. }
   error_$init_std_format (stream_$errout,  { Error output stream }
                           '?',             { Prefix character }
                           'PROG1',         { Command name }
                            5);             { Namelength }
```

Example 2-2. Formatting Error Messages with System Calls

```
{ Create a file. }
stream_$create (pathname,
                namelength,
                access,
                conc,
                stream_id,
                status);

IF status.all <> status_$ok THEN BEGIN
    { Print error message. }
     error_$std_format (status,
                        'Error creating file "%la" %$',
                        file_name,
                        name_length);

    { Invoke error handling procedure. }
    error_routine;
END;
        .
        .
        .
```

Example 2-2. Formatting Error Messages with System Calls (Cont.)

If the user attempts to open an existing file, this program produces the following error message:

?(format) Error creating file "file.dat" - file already exists (stream_$write specified on create) (stream manager/open)

2.6. Testing for Specific Errors

In some cases, you will wish to test for specific errors. A number of system calls return status codes that require special handling. The following is a nonexhaustive list of such status codes, and the calls that return them.

STREAM_$END_OF_FILE
: Returned by the STREAM_$GET calls when an end of file is encountered (for example, a CTRL/Z from a keyboard).

EC2_$WAIT_QUIT
: Returned by the EC2_$WAIT_SVC call when an asynchronous fault occurs while faults are inhibited.

PFM_$CLEANUP_SET
: Returned by the PFM_$CLEANUP call when a clean-up handler is successfully established.

The following program segment shows how a clean-up handler tests for the PFM_$CLEANUP_SET status code.

```
{ Clean-up handler code. }
status := PFM_$CLEANUP (handler_id); { Establish clean-up handler}

{ Check for est. status. }
IF (status.all <> PFM_$CLEANUP_SET) THEN BEGIN
        .
        .
        .
{ End of clean-up handler. }
ELSE BEGIN
```

When testing for a specific error from the STREAM subsystem, testing status.all is *not* sufficient. You must test two fields of the status record:

- Test the subsys field against the predefined value STREAM_$SUBS.
- Test the code field against the predefined error code.

The program segment in Example 2-3 shows a loop that reads records from a file. After each read, it tests for the STREAM_$END_OF_FILE error.

```
                .
                .
                .
{ Enter loop to get and print  records. }
WHILE (status.all = status_$ok) DO BEGIN

    { Get a record. }
    stream_$get_rec( stream_id,
                     addr(info_rec),
                     sizeof(info_rec),
                     retptr,
                     retlen,
                     seek_key,
                     status);
    { Test for EOF. }
    IF (status.code = stream_$end_of_file) AND
       (status.subsys = stream_$subs) THEN
        EXIT;
    IF (status.all <> status_$ok) THEN
        error_routine
    ELSE BEGIN
        { Assign returned pointer to buffer. }
        info_rec := retptr^;

        { Print the name and id fields. }
        writeln('name: ', info_rec.name:info_rec.namelen);
        writeln('id: ', info_rec.emp_id);
    END;  { if }
END;  { while }
```

Example 2-3. Testing for Specific STREAM Errors

2.6.1. Setting a Severity Level

In addition to exiting a program at the end of an error handling procedure, you may wish to set a severity level for your program, if your program:

- Is invoked by another program; for example, the Shell.
- Has a single, well-defined function.
- Is not interactive.

A **severity level** informs an invoking program of the completion status of an invoked program. You can use various features of the Shell, such as the ABTSEV command, to control the execution of Shell scripts based on the severity code. You set a severity level by calling PGM_$SET_SEVERITY. The error routine in Example 2-2 sets a severity level. See Chapter 3 for details about how to set a severity level.

2.7. Faults

While an error is detected by the algorithms of a system call and returned as a status code, a fault is detected (usually) by the hardware of the machine, and is not detected until the actual machine instructions are executed.

Depending on the exact nature of a fault, you may be able to "handle" the fault and continue processing. A fault that permits you to continue processing is referred to as **restartable**. (Restarting is highly application-dependent, and is beyond the scope of this manual.)

There are three ways to handle faults:

- Establishing clean-up handlers, described in Section 2.8.
- Establishing fault handlers, described in Section 2.9.
- Inhibiting asynchronous faults, described in Section 2.10.

The different types of faults you may encounter are described in this section. Every fault is either synchronous or asynchronous. Sections 2.7.1 and 2.7.2 describe synchronous and asynchronous faults, respectively.

Table 2-1 lists the predefined mnemonic constants for each of the faults that may be encountered on the system, along with a brief explanation of what causes the fault. These mnemonic constants are defined in the FAULT insert files, and are used by fault handlers to target specific faults.

Table 2-1. Summary of Faults

Fault	Explanation
FAULT_$ADDRESS_ERROR	Used odd address.
FAULT_$ILLEGAL_INST	Executed illegal instruction.
FAULT_$ZERO_DIVIDE	Divided by zero.
FAULT_$CHK_INST	CHK instruction trapped, index out of range?
FAULT_$TRAPV_INST	Arithmetic overflow occurred.
FAULT_$PRIV_VIOLATION	Privileged instruction violation.
FAULT_$ILLEGAL_SVC_CODE	Executed unrecognized SVC instruction.
FAULT_$ILLEGAL_SVC_NAME	Not currently used.
FAULT_$UNDEFINED_TRAP	Executed undefined TRAP instruction (6 thru 13).
FAULT_$UNIMPLEMENTED_INST	Executed unimplemented instruction.
FAULT_$PROT_VIOLATION	Protection boundary violation.
FAULT_$BUS_TIMEOUT	Bus time-out occurred.
FAULT_$ILLEGAL_USP	Invalid user stack pointer detected.
FAULT_$ECCC	Correctable memory error detected, (DN420, DN460, DN600, DN660 only).
FAULT_$ECCU	Uncorrectable memory error detected, (DN420, DN460, DN600, DN660 only).
FAULT_$QUIT	Executed process quit (CTRL/Q).
FAULT_$ACCESS_VIOLATION	Attempted to access protected memory or write read-only memory.
FAULT_$NOT_VALID	Hardware crash status (DN420, DN600 only).
FAULT_$NULLPROC_ONB	Hardware crash status (DN420, DN600 only).
FAULT_$DISPLAY_QUIT	OS-internal quit (with display return).
FAULT_$SINGLE_STEP	Executed instruction with trace bit on.
FAULT_$INVALID_USER_FAULT	Invalid user-generated fault.
FAULT_$PBU_USER_INT_FAULT	Fault in interrupt handler for PBU device.

Table 2-1. Summary of Faults (Cont.)

Fault	Explanation
FAULT_$STOP	Executed process stop instruction (dq -s).
FAULT_$BLAST	Executed process blast (dq -b).
FAULT_$CACHE_PARITY	PEB cache parity error detected.
FAULT_$WCS_PARITY	WCS parity error detected.
FAULT_$NOT_IMPLEMENTED	Issued unimplemented SVC instruction.
FAULT_$INVALID_STACK	Invalid stack format detected.
FAULT_$PARITY	Memory parity error detected.
FAULT_$INTERRUPT	Executed process interrupt.
FAULT_$WHILE_LOCK_SET	Fault occurred while resource lock(s) set.
FAULT_$SPURIOUS_PARITY	Spurious parity error detected.
FAULT_$FP_INEXACT	Floating point inexact result.
FAULT_$FP_DIV_ZERO	Floating point divide by zero.
FAULT_$FP_UNDFLO	Floating point underflow.
FAULT_$FP_OP_ERR	Floating point operand error.
FAULT_$FP_OVRFLO	Floating point overflow.
FAULT_$FP_BSUN	Floating point branch/set on an unordered condition.
FAULT_$FP_SIG_NAN	Floating point signaling not-a-number.
FAULT_$SUSPEND_PROC	Process suspend fault.
FAULT_$SUSPEND_PROC_KBD	Process suspend from keyboard.
FAULT_$SUSPEND_PROC	Process suspend due to background read.
FAULT_$SUSPEND_PROC	Process suspend due to background write.
FAULT_$CONTINUE_PROC	Process continue fault.
FAULT_$FAULT_LOST	Fault(s) lost; process suspended or inhibit count problem.
FAULT_$ILLEGAL_COPROC	Executed illegal coprocessor instruction.

2.7.1. Synchronous Faults

Synchronous faults occur as the result of an instruction executed by your program. The following two tables list specific types of synchronous faults and whether or not they are restartable. Table 2-2 lists program faults. Program faults are caused directly by an action of your program.

Table 2-2. Synchronous Program Faults

Program Faults	Description
Unimplemented instruction	Restartable.
Odd address error	Not restartable. (Typically caused by a bad pointer.)
Reference to an invalid address	Not restartable.
Access violation	Not restartable.
Reference to an unresolved global	Not restartable.
Guard fault	Restartable.

Table 2-3 lists system faults. System faults are triggered by a program instruction, but occur because of a failure on the part of the system.

Table 2-3. Synchronous System Faults

System Faults	Description
Network failure	Not restartable. (Typically, occurs during paging across the network.)
Disk full	Not restartable. (Use the Alarm Server to avoid disk full errors.)
Disk error	Not restartable.

2.7.2. Asynchronous Faults

Asynchronous faults are produced from outside of your program. They can occur at any point in your program and are unrelated to anything your program did. A common example is the "quit fault," caused by the Display Manager's DQ command (usually when someone types CTRL/Q to stop a program).

You may choose to handle asynchronous faults, or you may choose to inhibit the delivery of asynchronous faults. Section 2.10 describes how to inhibit asynchronous faults.

2.8. Handling Faults with Clean-Up Handlers

Typically, you use a clean-up handler in programs when you wish to deal with faults by terminating normal processing. A **cleanup handler**, like its name implies, is used to *clean up* a process before the program exits. Before exiting, the clean-up handler might restore disk files or in-memory tables to a known or stable state, or restore other things the program has changed. When a fault occurs, the process fault manager *automatically* calls the PFM_$SIGNAL system call as part of the fault handling process. PFM_$SIGNAL invokes the clean-up handler on the top of the stack, passing the fault status.

You may also use clean-up handlers to let the program continue processing after a fault occurs. However, a clean-up handler effects a *nonlocal GOTO* when a fault occurs. Control passes to the clean-up handler code, and the context in which the fault occurred is destroyed, so it is not possible to return to the point in the code at which the fault occurred. If you choose to continue processing after handling a fault, control passes to the point after the clean-up code.

Note that there is a built-in clean-up handler. This handler is established when PGM_$INVOKE was called to invoke your program. The built-in handler always closes any files that are still open and returns control to the invoking program, such as the Shell.

Because of the way in which the clean-up handlers are invoked, you should not establish clean-up handlers to work across program levels. That is, if you perform an operation that requires clean-up in a subroutine or function, the handler should be established and released within the subroutine or function.

Once a clean-up handler handles a fault, the process fault manager releases the handler; it will not handle future faults unless you re-establish it. Re-establishing clean-up handlers is described in Section 2.8.4.3.

Asynchronous faults are inhibited during the execution of a clean-up handler, so that the program cannot be interrupted while it is trying to clean up.

2.8.1. Establishing a Clean-Up Handler

To establish a clean-up handler:

1. Call PFM_$CLEANUP. The initial call to PFM_$CLEANUP returns a status of PFM_$CLEANUP_SET (indicating that the handler has been established). It also returns a unique identifier for the handler (referred to as the handler-ID) that permits you to identify specific clean-up handlers when using more than one.

2. Construct an IF-THEN-ELSE block that tests the status returned by PFM_$CLEANUP. If the status is equal to PFM_$CLEANUP_SET, branch to the beginning of normal operations. If the status is not equal to PFM_$CLEANUP_SET, a fault is assumed and the clean-up operations should be performed.

3. Write the clean-up operation. What operations are performed as part of the clean-up depends on what the program does. If files are opened and created in the program, you may want to close or delete them in the clean-up handler to ensure a stable state.

If your program contains a clean-up handler, it is invoked when a fault occurs *or* when PFM_$SIGNAL is invoked. (PFM_$SIGNAL invokes the topmost clean-up handler on the stack (if there is one), passing it a status code; it can be called from any point in a program.) At that point, control immediately returns to the place in your program where you call PFM_$CLEANUP. In this case, the status test fails and the clean-up code is executed.

The program segment in Example 2-4 creates a file and performs I/O on it. It establishes a clean-up handler that deletes the file and exits, if a fault occurs during the processing of the file. Note that the variable stream_open is used to indicate that a stream has been opened to the file. The clean-up handler checks the state of this variable to determine whether it should delete the file. This prevents the handler from attempting to delete the file if a fault occurs before the file is created.

```
PROGRAM pfm_clean_up (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

VAR
  status       : status_$t;
  stream_open  : boolean;  { State variable }
  count        : integer;  { VFMT parameter }
  { $CREATE variables }
  pathname     : name_$pname_t;
  namelength   : integer;
  stream_id    : stream_$id_t;
  { $CLEANUP variable }
  handler_id   : pfm_$cleanup_rec;

PROCEDURE error_routine;   { for error handling }
BEGIN
    error_$print( status );
END;  { error_routine }

BEGIN  { Main Program }

    { Initialize state variable. }
    stream_open := FALSE;       { Not open yet }

    { Clean-up handler code. }
    status := pfm_$cleanup (handler_id); { Establish clean-up handler }

    { Check for established status. }
    IF (status.all <> pfm_$cleanup_set) THEN BEGIN

        { Delete file if open while fault occurs. }
        IF stream_open THEN
            stream_$delete (stream_id,
                            status);
        pgm_$exit;
    END;      { of clean-up handler }
```

Example 2-4. Establishing A Clean-Up Handler

```
    { Begin normal operations. }

    { Get the filename. }
    writeln ('Input pathname: ');
    readln (pathname);

    { Calculate namelength. }
    namelength := sizeof(pathname);
    WHILE (pathname[namelength] = ' ') AND
          (namelength > 0 ) DO
      namelength := namelength - 1;

    stream_$create (pathname,
                    namelength,
                    stream_$write,              { Access }
                    stream_$controlled_sharing, { Concurrency }
                    stream_id,
                    status);

    IF status.code <> status_$ok THEN
        error_routine;

    { Set state variable. }
    stream_open := TRUE;  { File is open }

    { Get the input. }
          .
          .
          .
    { Finish processing the file. }
    { Release the clean-up handler. }
    pfm_$rls_cleanup (handler_id,
                      status);
          .
          .
          .
END. { pfm_clean_up }
```

Example 2-4. Establishing A Clean-Up Handler (Cont.)

2.8.2. Releasing a Clean-Up Handler

Note that the program segment in Example 2-4 releases the handler when it finishes processing the file. When a clean-up handler is no longer needed, it should be released. Releasing a handler removes it from the stack that the process fault manager uses to keep track of handlers.

You release handlers to prevent invoking clean-up code when it is not appropriate. Often, a clean-up handler applies to only one section of a program, and should not take effect if a fault occurs later in the program. For instance, in Example 2-4, the file might have been properly processed and closed, leaving it in a stable state. Yet, had the handler not been released, a fault might have occurred before the program completed, and the file would be needlessly deleted.

To release a clean-up handler, call PFM_$RLS_CLEANUP, specifying the handler ID of the handler you want to release. The call to PFM_$CLEANUP returns the handler ID when you establish the handler.

A procedure, function, or subroutine must release all the clean-up handlers it established before returning to its caller.

After a clean-up handler handles a fault, the process fault manager releases it, unless it is explicitly re-established. A clean-up handler that has been released by the process fault manager may be placed back on the stack by *re-establishing* it. See Section 2.8.4.3.

```
NOTE: When a handler is released, all handlers established more recently than
      that handler are also released.
```

2.8.3. Multiple Clean-Up Handlers

More than one clean-up handler can be in effect at once. The process fault manager invokes clean-up handlers on a last-in-first-out (LIFO) basis. The last clean-up handler that gains control is the built-in clean-up handler (as it is the first to be established).

When you have a number of clean-up handlers, it is important that each handler be invoked only when appropriate. One way to help ensure this is to release clean-up handlers when you no longer need them, as stated above. In addition, you may wish to use *state variables* to ensure that a handler is not invoked *before* it is needed.

For example, if you establish a clean-up handler to clean up a file that you modify, declare a Boolean variable that you set to TRUE when you open the file. Write the clean-up handler so that it tests the Boolean before trying to clean up the file. If the file has been opened, the handler cleans up. If the file has not been opened, the handler does not attempt the clean-up. Example 2-4 uses the variable stream_open as a state variable.

2.8.4. Exiting a Clean-Up Handler

There are four ways to exit a clean-up handler:

- Resignaling passing the fault status.
- Resignaling passing a severity level.
- Re-establishing the handler and returning to the program.
- Returning to the program.

2.8.4.1. Resignaling Passing the Fault Status

Resignaling is the act of passing the signaled fault to the next handler in the process fault manager's stack. Typically, a handler resignals a fault when you want to invoke a number of (or all) established clean-up handlers.

To resignal a fault status, a handler calls PFM_$SIGNAL, specifying the status returned to it by the PFM_$CLEANUP call.

If you resignal and your program has no more clean-up handlers, control passes to the built-in clean-up handler, and eventually your program exits to the invoking program. When this occurs, your program returns the fault status as its severity code.

2.8.4.2. Resignaling Passing a Severity Level

If another program invokes your program, the invoking program may expect your program to return a severity level instead of a fault status. Every program starts with the severity level set to PGM_$OK (successful completion). When a fault occurs, you may change the severity level by calling PGM_$SET_SEVERITY.

To resignal a fault by passing a severity level:

1. Call PGM_$SET_SEVERITY to set the severity to the chosen level.

2. Call PGM_$EXIT.

PGM_$EXIT resignals the next clean-up handler, but instead of passing the fault status code, it passes a status code that translates to the severity level.

The following is a clean-up handler that sets the severity level to PGM_$ERROR, then resignals.

```
{ Clean-up handler code. }
status := pfm_$cleanup (handler_id); { Establish clean-up handler }

{ Check for est. status. }
IF (status.all <> pfm_$cleanup_set) THEN BEGIN
      { Delete file if fault occurs. }
      stream_$delete (stream_id,
                      status);
      pfm_$set_severity(pgm_$error);
      pgm_$exit;
      END    { of clean-up handler }
```

See Chapter 3 for more information about setting severity levels.

2.8.4.3. Re-establishing the Handler and Returning to the Program

Once a clean-up handler is invoked, it is released and will not be invoked again, unless you specifically re-establish it.

You re-establish a handler if you are restarting after the fault and there will still be a need for the handler. Consider, as an example, a program that processes files based on commands that the user input. This program needs a clean-up handler to clean up a file if a fault occurs, but can easily continue processing by getting the next command. The program can simply establish one handler that re-establishes itself.

To re-establish a clean-up handler, call PFM_$RESET_CLEANUP, specifying the handler ID. When you re-establish a handler, fault handling stops (no other handlers on the stack are invoked). The re-established handler is now the most-recently-established clean-up handler and will be the first clean-up handler to handle the next fault. The program can now continue running, but cannot return directly to the point where the fault occurred.

The following is a clean-up handler that resets itself and re-enables asynchronous faults.

```
{ Clean-up handler code. }
status := pfm_$cleanup (handler_id); { Establish clean-up handler }

{ Check for est. status. }
IF (status.all <> pfm_$cleanup_set) THEN BEGIN
      { Delete file if fault occurs. }
      stream_$delete (stream_id,
                      status);
      pfm_$reset_clean_up (handler_id,
                           status);
      END    { of clean-up handler }
```

2.8.4.4. Returning to the Program

In some cases, you may wish to simply handle a fault and return to the program, without re-establishing a clean-up handler. One example is a program that is performing a number of loosely connected tasks. Your program may abort one task and continue by processing the next task on the list.

No special action is required to return to the program. However, because asynchronous faults are disabled when a clean-up handler is invoked, you should re-enable them before returning. To re-enable asynchronous faults, call PFM_$ENABLE.

The following is a clean-up handler that re-enables asynchronous faults and returns to the program.

```
{ Clean-up handler code. }
status := pfm_$cleanup (handler_id); { Establish clean-up handler. }

{ Check for est. status. }
IF (status.all <> pfm_$cleanup_set) THEN BEGIN
      { Delete file if fault occurs. }
      stream_$delete (stream_id,
                      status);
      pfm_$enable;
      END    { of clean-up handler }
```

2.8.5. Handling Errors With Clean-Up Handlers

You can also use clean-up handlers to handle error conditions. However, unlike fault conditions, error conditions do *not* automatically call PFM_$SIGNAL to pass to a clean-up handler.

To invoke a clean-up handler for an error condition, your program must:

- Detect the error condition.
- Call PFM_$SIGNAL, passing the error status to the clean-up handler on the top of the stack.

The program in Example 2-5 creates a file and calls a procedure to write to it. The main program declares a clean-up handler that deletes the file before exiting. If an error occurs while writing data to the file the procedure invokes the clean-up handler by explicitly calling PFM_$SIGNAL.

```
PROGRAM pfm_clean_error (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  status      : status_$t;
  stream_open : boolean;  { State variable }
  count       : integer;  { VFMT parameter }

  { $CREATE variables }
  pathname   : name_$pname_t;
  namelength : integer;
  stream_id  : stream_$id_t;

  { $CLEANUP variable }
  handler_id : pfm_$cleanup_rec;

{************************************************************}

PROCEDURE error_routine; { for error handling }
BEGIN
    error_$print( status );
END;  { error_routine }

{************************************************************}
{ Procedure to write to file                                 }
{************************************************************}

PROCEDURE write_to_file (str_id : stream_$id_t);

VAR
  line      : ARRAY[1..80] OF char;
  seek_key  : stream_$sk_t;
  buflen    : integer32;

BEGIN
    { Get a line of input. }
    writeln ('Input data (or CTRL/Z to stop):');
    WHILE NOT eof DO
    BEGIN
        readln(line);
        buflen := SIZEOF(line);
        WHILE (line[buflen] = ' ') AND (buflen > 0) DO
            buflen := buflen - 1;

        { Terminate line with newline character. }
        buflen := buflen + 1;
        line[buflen] := CHR(10);
```

Example 2-5. Invoking a Clean-Up Handler for an Error

```
        { Write the line to a file. }
        stream_$put_rec ( str_id,
                          ADDR(line),
                          buflen,
                          seek_key,
                          status);
        { Invoke clean-up handler if error occurs. }
        IF status.code <> status_$ok THEN
           pfm_$signal(status);

        writeln ('Record written');
        writeln ('Input more info (or CTRL/Z to stop):');
    END;{while}

END; { write_to_file }

{*************************************************************}

BEGIN  { Main Program }

    { Initialize state variable. }
    stream_open := FALSE;                { Not open yet }

    { Clean-up handler code }
    status := pfm_$cleanup (handler_id); { Establish clean-up handler }

    { Check for established status. }
    IF (status.all <> pfm_$cleanup_set) THEN BEGIN

        { Delete file if open while fault occurs. }
        IF stream_open THEN
            stream_$delete (stream_id,
                            status);
        writeln ('Output file deleted - write error occurred');
        pgm_$exit;
    END;     { of clean-up handler }

    { Begin normal operations. }

    { Get the filename. }
    writeln ('Input pathname of file to be written: ');
    readln (pathname);

    { Calculate namelength. }
    namelength := sizeof(pathname);
    WHILE (pathname[namelength] = ' ') AND
          (namelength > 0 ) DO
      namelength := namelength - 1;

    stream_$create (pathname,
                    namelength,
                    stream_$write,              { Access }
                    stream_$controlled_sharing, { Concurrency }
                    stream_id,
                    status);
```

Example 2-5. Invoking a Clean-Up Handler for an Error (Cont.)

```
    IF status.code <> status_$ok THEN
        error_routine;

    { Set state variable. }
    stream_open := TRUE;  { File is open. }

    { Call procedure to write to the file. }
    write_to_file (stream_id);

    { Finished processing the file. }
    { Release the clean-up handler }
    pfm_$rls_cleanup (handler_id,
                      status);
END. { pfm_clean_error }
```

Example 2-5. Invoking a Clean-Up Handler for an Error (Cont.)

2.9. Handling Faults with Fault Handlers

A **fault handler** is a procedure that is called when a fault occurs; unlike a clean-up handler, it is capable of returning to the point at which the fault occurred.

A fault handler might handle faults where you want to respond to the fault by taking some corrective action and continuing normal processing.

2.9.1. Establishing a Fault Handler

To establish a fault handler:

1. Write a function that performs the actual fault handling.
2. Call PFM_$ESTABLISH_FAULT_HANDLER to establish the function as a fault handler.

2.9.1.1. Writing the Fault-Handling Function

You must write a fault handler as a function.

Each fault-handling function takes one input parameter, the fault record. The **fault record** is a data type, in PFM_$FAULT_REC_T format. One field of this record contains the fault status. When a fault occurs, the process fault manager loads this record and invokes the handler.

The value that a fault-handling function returns determines the action taken after the fault is handled. The return value for a fault handler must be in PFM_$FH_FUNC_VAL_T format (a 2-byte integer), and must be set to one of the following two predefined values:

PFM_$CONTINUE_FAULT_HANDLING
: Indicates that the program should invoke any other established fault handlers. If no more handlers exist, clean-up operations are invoked.

PFM_$RETURN_TO_FAULTING_CODE
: Indicates that control should return to the program. No further fault handling is performed. The program restarts after the instruction that took the fault.

In Pascal, the fault handling function must be in a Pascal MODULE (as opposed to a PROGRAM). The call that establishes the fault handler passes the system the address of the function -- this cannot be done from within a Pascal PROGRAM.

Example 2-6 is a module in which a fault-handling function named "zero_fault_handler" is declared. The return value is set to PFM_$CONTINUE_FAULT_HANDLING, specifying that any other established handlers should be invoked.

```
MODULE pgm_zero_handler; (input,output);

{ This is a fault handling function that prints }
{ a line and continues to fault handle.         }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pfm.ins.pas';

FUNCTION zero_fault_handler (IN f_status : pfm_$fault_rec_t { Fault record }
                            ): pfm_$fh_func_val_t ;         { Return value }

BEGIN
    { Write a message to the error log. }
    error_$print (f_status.status);

    { Load the return value. }
    zero_fault_handler := pfm_$continue_fault_handling;

END; { zero_fault_handler }
```

Example 2-6. Writing a Fault-Handling Function

2.9.1.2. Establishing the Function as a Handler

Before a fault-handling function can be used by a program, it must be established as a fault handler. To establish a function as a fault handler, call the PFM_$ESTABLISH_FAULT_HANDLER function. The following is the syntax for PFM_$ESTABLISH_FAULT_HANDLER:

```
handler_id := pfm_$establish_fault_handler (target_fault,
                                             type_options,
                                             address_of_function,
                                             status);
```

You pass PFM_$ESTABLISH_FAULT_HANDLER three input parameters:

- A target fault, as a 4-byte integer.

- An option describing the type of handler you are establishing, in PFM_$FH_OPT_SET_T format.

- The address of the fault handling function, in PFM_$FAULT_FUNC_P_T format.

PFM_$ESTABLISH_FAULT_HANDLER uses this information to establish the handler and returns a handler ID that uniquely identifies the handler. The call also returns a completion status.

You can specify the address of the function using the ADDR extension to DOMAIN Pascal, or the IADDR special function of DOMAIN FORTRAN.

2.9.1.3. Setting Target Faults

PFM_$ESTABLISH_FAULT_HANDLER's target fault parameter permits you to specify the fault(s) to which you want a handler to respond. You can specify one specific fault, a group of faults, or all faults. The target fault parameter expects a 4-byte integer value.

- To specify a specific fault, simply specify the parameter to be the specific fault status code.

- To specify all the faults in a DOMAIN module, specify any status code returned by that module, with the fault code field set to zero. The following program example sets the target fault to be all faults in the SMD module:

```
VAR
   target_fault : status_$t;
   .
   .     { Declare other variables. }
   .
BEGIN
    { Load the target fault. }
    target_fault := smd_$illegal_unit;
    target_fault.code := 0;

    { Establish the fault handler. }
    handler_id := pfm_$establish_fault_handler (target.all, {integer32}
                                                 options,
                                                 ADDR(my_fault_handler),
                                                 status);
```

- To specify all faults, specify the predefined constant PFM_$ALL_FAULTS.

2.9.1.4. Specifying Handler Types

You can establish fault handlers to be of three types. Table 2-4 lists them.

Table 2-4. Types of Fault Handlers

Fault Handler	Description
Default	By default, if a number of fault handlers are responding to a fault, they are invoked in reverse order of establishment (LIFO) and applies to the program level in which it is established. To specify a default handler, specify the null set.
Backstop handlers	If you specify a handler to be a backstop handler, the process fault manager does not invoke it until all the nonbackstop handlers have been invoked.
Multilevel handlers	If you specify a handler to be a multilevel handler, it applies to the program level at which it is established and all subordinate program levels. This means that the fault handler will be executed for the program that establishes the fault handler and for any programs that the program invokes (even though they do not establish a fault handler).

Backstop and multilevel types are not exclusive of each other; a handler can be both a backstop and multilevel fault handler.

The program segment in Example 2-7 establishes the function shown in Example 2-6 as a default-type handler that responds to the FAULT_$ZERO_DIVIDE fault. Note that the program includes the FAULT insert file that defines this fault.

```
PROGRAM pgm_divide (input, output);

{ Program to divide two numbers. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/fault.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  number1     : integer;
  number2     : integer;

  status      : status_$t;
  handler_id  : pfm_$fh_handle_t;
```

Example 2-7. Establishing a Fault Handler

```
{ Declare external fault handling function. }
FUNCTION zero_fault_handler (IN f_status : pfm_$fault_rec_t
                             ): pfm_$fh_func_val_t; EXTERN;

BEGIN  { Main Program }

    { Establish the zero divide handler. }
    { Load the target fault -- first parameter. }
    handler_id := pfm_$establish_fault_handler (fault_$zero_divide,
                                                 [],         { Default type }
                                                 ADDR(zero_fault_handler),
                                                 status);

    IF (status.all <> status_$ok) THEN
        error_$print (status);
            .
            .
            .
```

Example 2-7. Establishing a Fault Handler (Cont.)

2.10. Inhibiting Asynchronous Faults

During part or all of your program, you can inhibit asynchronous faults. Inhibiting asynchronous faults defers the effect of the CTRL/Q key in stopping the program. This is appropriate when there are intervals during which your program must not be interrupted. For instance, your program may perform some I/O that would be left in an inconsistent state if the user were allowed to interrupt execution. However, it is good programming practice to only inhibit asynchronous faults during these critical intervals, so that a user may terminate the program at some point, if necessary.

To inhibit asynchronous faults, call PFM_$INHIBIT. This call has no parameters.

To re-enable asynchronous faults, call PFM_$ENABLE. This call also has no parameters.

If a fault occurs while asynchronous faults are inhibited, the system holds the fault for delivery when faults are re-enabled. However, the system will only hold one fault; all others are ignored.

The operating system keeps track of inhibits by incrementing and decrementing an **inhibit count**. Asynchronous faults are only delivered when the inhibit count is 0. Each time an inhibit occurs (either explictly called by you, or implicitly called by the system - as during a clean-up handler) the count is incremented. It is decremented any time a call is made that re-enables asynchronous faults, such as PFM_$ENABLE. This is why clean-up handlers that return to the invoking program must call PFM_$ENABLE before returning.

Inhibiting asynchronous faults has no effect on the delivery of synchronous faults.

Chapter 3
Managing Programs

Programs are normally divided into a number of smaller program units, which perform specific tasks. Program units may take three forms:

1. Subroutines, procedures, and functions that you write.

2. DOMAIN system calls. System calls are procedures and functions that can be called to perform specific predefined tasks. The *DOMAIN System Call Reference* alphabetically lists all the available system calls and describes what each of them does.

3. Other programs. DOMAIN permits you to invoke other programs from within your program. You can invoke other programs you have written, or you can invoke system-provided programs (i.e., DOMAIN Shell commands).

This chapter describes how to invoke programs with the PGM system calls and how to obtain process information with the PROC and PM system calls.

3.1. System Calls, Insert Files, and Data Types

To invoke and manage programs, use system calls with the prefix PGM. In order to use PGM system calls, you must include the appropriate insert file in your program. The PGM insert files are:

/SYS/INS/PGM.INS.C	for C programs.
/SYS/INS/PGM.INS.FTN	for FORTRAN programs.
/SYS/INS/PGM.INS.PAS	for Pascal programs.

To obtain process information, use the system calls with the prefix PROC1, PROC2, or PM, depending on what inmformation you want. You must also include the appropriate insert file. The insert files are:

/SYS/INS/prefix.INS.C	for C programs.
/SYS/INS/prefix.INS.FTN	for FORTRAN programs.
/SYS/INS/prefix.INS.PAS	for Pascal programs.

where prefix is the desired subsystem prefix.

This chapter is intended to be a guide for performing certain programming tasks; the data type and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference*.

3.2. Invoking External User Programs

Invoking programs from within a program avoids having to duplicate the work of existing programs. It also provides a way of performing concurrent processing.

To invoke the execution of another program, use PGM_$INVOKE. PGM_$INVOKE permits you to pass arguments and stream connections to the invoked program. How to pass arguments to an invoked program is described in Section 3.3. How to pass streams to an invoked program is described in Section 3.6.

PGM_$INVOKE returns two parameters: the process handle and the completion status. The **process handle** uniquely identifies the invoked program and is used as an input parameter to other system calls; for example, PGM_$PROC_WAIT. This completion status is slightly different from the completion status of other system calls because it is interpreted differently depending on the mode in which a program is invoked. How to interpret the completion status is described, along with the invoke modes, in the following sections.

When you call PGM_$INVOKE, you have three options for the mode in which the invoked program will execute:

Wait mode — The program executes as a separate program within the same process as the invoking program. The invoking program 'waits' until the invoked program is completed before resuming execution. (Described in Section 3.2.1.)

Default mode — The program executes as a separate process that communicates its termination status to the invoking program. (Described in Section 3.2.2.)

Background mode — The program executes as a separate process that runs to termination independently of the invoking process. (Described in Section 3.2.3.)

The mode in which you choose to run a program depends on the task performed by the program.

3.2.1. Invoking a Program in Wait Mode

To invoke a user program in wait mode, call PGM_$INVOKE, with the mode option set to PGM_$WAIT. When you invoke a program this way, the invoking program executes the program and waits for it to complete before continuing. In this respect, calling PGM_$INVOKE with the WAIT option is similar to calling a subroutine.

Executing a program within your own process avoids the overhead associated with process creation.

The DOMAIN Shell is an example of using INVOKE with the PGM_$WAIT option. Each Shell command is a program, and the options to the command are arguments. The Shell invokes the specified program passing any arguments, and waits for the program to complete.

You may also wish to invoke an existing Shell command from within a program. The program segment in Example 3-1 invokes the "date" Shell command, using PGM_$WAIT mode. Note that the invoking program passes the invoked program the four standard streams. It is good programming practice to pass an invoked program the standard streams. Section 3.6 describes how to pass streams. The "date" program writes the date to the standard output stream.

```
PROGRAM pgm_shell;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  handle : pgm_$proc;
  status : status_$t;

  {declare and load the standard streams}
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;

BEGIN
    pgm_$invoke('/com/date',
                9,
                0, 0,       { no args }
                4,
                connv,      { std. streams }
                [pgm_$wait],
                handle,
                status) ;
    check_status;
END.
```

Example 3-1. Invoking an Existing Shell Command

A program that you invoke in wait mode is said to be running at a higher program level. The invoking program is at program level n, while the invoked program is at program level n+1. If the invoked program were, in turn, to invoke a third program, the third program would be at program level n+2, and so on. The context of an invoking program level is preserved while an invoked program is executing. The context is restored when an invoked program terminates.

3.2.1.1. Setting Severity Levels

Typically, an invoked program returns a severity level when returning from a higher program level. A **severity level** indicates the completion status of an invoked program. To set a severity level, call PGM_$SET_SEVERITY, passing it one of the predefined severity levels listed in Table 3-1. Then call PGM_$EXIT to exit the current program level.

For a program invoked in wait mode, PGM_$EXIT returns the severity level in the status of the PGM_$INVOKE call. Of course, the return status may also indicate that the PGM_$INVOKE call failed to invoke the specified process.

Using the severity levels requires coordination between the invoking program and the invoked

Table 3-1. Severity Levels

Severity Level	Description
PGM_$OK	The program completed successfully and performed the requested action. This is the default severity level.
PGM_$TRUE	The program completed successfully; its purpose was to test a condition, the value of that condition was TRUE.
PGM_$FALSE	The program completed successfully; its purpose was to test a condition, the value of that condition was FALSE.
PGM_$WARNING	The program completed successfully and performed the requested action. However, an unusual (but nonfatal) condition was detected.
PGM_$ERROR	The program could not perform the requested action because of syntactic or semantic errors in the input. The output is structurally sound, however.
PGM_$OUTPUT_INVALID	The program could not perform the requested action because of syntactic or semantic errors in the input, and the output is not structurally sound.
PGM_$INTERNAL_FATAL	The program detected an internal fatal error and ceased processing. The state of the output is neither defined nor guaranteed.
PGM_$PROGRAM_FAULTED	The program detected a fault.

program. An invoked program may interpret status codes as *belonging* to a specific severity level. However, this interpretation is strictly determined by how the invoked program is written. For example, one program may interpret a STREAM_$INVALID_PATHNAME code as an error, while another may interpret it as a warning.

Depending on the severity level returned from a program, an invoking program may continue processing, take an appropriate action, or signal the severity level and exit.

Example 3-2 contains two programs: PGM_INVOKE.PAS and PGM_OPEN.PAS. INVOKE invokes the program OPEN in wait mode. OPEN opens a file and sets the severity level to PGM_$ERROR, if any status other than STATUS_$OK is returned. INVOKE signals the error and exits. Note that INVOKE's error-handling routine distinguishes between an error and a warning so that other programs it invokes may return a warning severity.

```
PROGRAM pgm_invoke;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  handle : pgm_$proc;
  status : status_$t;

  { declare and load the standard streams }
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        CASE status.all OF
            pgm_$error   : writeln ('Invoked program ended with error status');
            pgm_$warning : writeln ('Invoked program ended with warning status');
        END; {case}
        pgm_$exit;
    END; {if}
END; {procedure}

BEGIN
    pgm_$invoke('pgm_open.bin',
                12,
                0, 0,        {no arguments}
                4,
                connv,       {std. streams}
                [pgm_$wait],
                handle,
                status) ;
    check_status;
END.    {pgm_invoke}

           {****}

PROGRAM pgm_open (input,output);

%include '/sys/ins/base.ins.pas';
%include '/latest/us/ins/pgm.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  status     : status_$t;
  pathname   : name_$pname_t;
  namelength : integer;
```

Example 3-2. Returning a Severity Level from an Invoked Program

```
{ $open variable }
  stream_id   : stream_$id_t;

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        pgm_$set_severity (pgm_$error);
        pgm_$exit;
    END;
END;

BEGIN

{ open the file }
    stream_$open ('file.out',
                  9,
                  stream_$read,                {access}
                  stream_$controlled_sharing, {concurrency}
                  stream_id,
                  status);
    check_status;

END.
```

Example 3-2. Returning a Severity Level from an Invoked Program (Cont.)

3.2.2. Invoking a Program in Default Mode

To invoke a user program in default mode, call PGM_$INVOKE, with the mode option set to a null parameter. When you invoke a program this way, the invoking program creates a new process in which to run the program. A default mode process communicates its termination status to the invoking program through the PROC_$WAIT system call.

When a process invokes another process, the invoking process is referred to as the **parent process**, and the invoked process is referred to as the **child process**. Executing a program in a child process is useful if you wish to perform concurrent processing or if your program requires a large amount of address space (each process gets its own address space).

There are a number of things that should be considered before invoking a program in a child process:

- Creation of a new process is more expensive in terms of processor overhead. Unless you need the additional address space or are performing concurrent processing, it is recommended that you invoke programs in wait mode.

- A child process has its own process address space. This permits you the advantage of more address space. However, because private libraries are stored in the parent's address space, the child process has no access to the private libraries loaded in the parent process.

- A child process inherits some environment from the parent. A child process inherits the working directory of its parent, and also inherits any stream locks its parent may have.

- A parent process can pass any streams it holds to a child process, with the exception of magtape streams. It is a good practice to always pass the standard streams to a child process. Section 3.6 describes how to pass streams.

- Only some operations taken by a child process are permanent. For example, if a child process creates a file, the file exists even after the process terminates. However, if a child process performs a GPR_$INIT to initialize the graphics environment, when the child process terminates, the program exits the graphics environment, even if the invoked program does not call GPR_$TERMINATE. (This is true of *all* invoked programs.)

3.2.2.1. Waiting for a Child Process

If you are performing concurrent processing, you may wish to wait for a child process to complete before executing a specific piece of a program. For example, you may wish to add the results of calculations performed by both the parent and child processes.

There are two ways to wait for completion of a child process:

- Waiting on a process eventcount, using PGM_$GET_EC.

- Calling PGM_$PROC_WAIT.

The PGM_$GET_EC call permits you to get a process eventcount that is advanced when the process terminates. Generally speaking, you cannot depend on the actual value of an eventcount. However, you can depend on the value of the process eventcount. When a process is invoked, its eventcount value is set to 0. When a process terminates, its eventcount value is set to 1. These are the only two values a process eventcount can have. Because of this, you can explicitly set the satisfaction values of the process eventcounts to 1.

By using this call in conjunction with the system calls EC2_$READ and EC2_$WAIT, a parent process can wait for the completion of a child process (or a list of eventcounts). For general information about using eventcounts, see Chapter 6 of this manual.

PGM_$PROC_WAIT waits for a specified child process to terminate, and returns its completion status. (Typically, a child process returns severity levels in the same way that a program invoked in wait mode does.) PGM_$PROC_WAIT takes the process handle as an input parameter, and returns the completion status of the invoked process. If the child process has not completed execution at the time of the PROC_WAIT call, execution of the parent process suspends until a completion status is available. The process handle is only valid between the time a default mode process is invoked and the time the PGM_$PROC_WAIT mode completes.

A certain amount of resources in a parent process are used to keep track of a child process. When a call to PGM_$PROC_WAIT is completed, those resources are released. If you invoke a number of child processes without ever calling PROC_WAIT, the parent process may run out of resources. Once a child process has completed, you should call PGM_$PROC_WAIT to release these resources, whether you are interested in its completion status or not. That is, if you wait on a process using an eventcount, you must still call PGM_$PROC_WAIT.

If you are NOT interested in when or how an invoked program completes, invoke it using background mode (see Section 3.2.3).

The program in Example 3-3 invokes two child processes and gets an eventcount for each one. It then waits for each to complete, and processes the results. (Assume that the programs communicate results by mapping files -- see the *Programming With System Calls for Interprocess Communication* manual for information about mapping files.)

When the child processes terminate, their resources are released with a call to PGM_$PROC_WAIT. Note that handling process eventcounts differs from other eventcounts in the following ways:

- You explicitly initialize the eventcount satisfaction (trigger) value to 1. An eventcount of 1 indicates that a process has terminated.

- When you release the resources of the terminated process, its process eventcount (and the eventcount pointer) becomes invalid. This requires that you create a valid eventcount and eventcount pointer to take its place in the eventcount pointer array, while you wait for the other eventcounts to be satisfied. Otherwise, the EC2_$WAIT call will reference an illegal address. To do so, declare the replacement eventcount to be a variable in EC2_$EVENTCOUNT format, and load it with a valid eventcount by calling EC2_$INIT. In the example, this eventcount is the variable "replace_ec".

- You explicitly set the created eventcount value to 1. This guarantees that the eventcount will not be selected again.

- You replace the invalid pointer in the eventcount pointer array with a pointer to the eventcount you created.

```
PROGRAM pgm_ec;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ec2.ins.pas';

CONST
  calc1_ec = 1;
  calc2_ec = 2;

VAR
  ec2_ptr      : array [1..2] of ec2_$ptr_t;
  ec2_val      : array [1..2] of integer32;
  replace_ec   : ec2_$eventcount_t;
  which        : integer;
  status       : status_$t;
  dead_count   : integer;
  handle1      : pgm_$proc;
  handle2      : pgm_$proc;
```

Example 3-3. Using an Eventcount to Wait for a Child Process

```
  {declare and load the standard streams}
  connv   : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;

BEGIN

    {invoke 1st process}
    pgm_$invoke('calc1.bin', {program name}
                9,           {namelength}
                0, 0,        {no args}
                4,           {number of streams}
                connv,       {std. streams}
                [],          {default mode}
                handle1,     {process handle}
                status) ;
    check_status;

    {invoke 2nd process}
    pgm_$invoke('calc2.bin',
                9,
                0, 0,
                4,
                connv,
                [],
                handle2,
                status) ;
    check_status;

    {get ec for 1st process}
    pgm_$get_ec (handle1,          {process handle}
                 pgm_$child_proc,  {ec key}
                 ec2_ptr[calc1_ec],{ec_ptr}
                 status);
    check_status;

    {get ec for 2nd process}
    pgm_$get_ec (handle2,          {process handle}
                 pgm_$child_proc,  {ec key}
                 ec2_ptr[calc2_ec],{ec_ptr}
                 status);
    check_status;

    {map results files}
             .
             .
             .
```

Example 3-3. Using an Eventcount to Wait for a Child Process (Cont.)

```
{initialize the replacement event count}
ec2_$init (replace_ec);

{initialize counter}
dead_count := 0;

{initialize satisfaction values to 1}
ec2_val[calc1_ec] := 1;
ec2_val[calc2_ec] := 1;

{ NOW GO INTO A LOOP PROMPTING FOR INPUT }
REPEAT
    {determine which event count reaches satisfaction first}
    which := ec2_$wait (ec2_ptr,    {ec pointer array}
                        ec2_val,    {ec value array}
                        2,          {number of ec's}
                        status);
    IF status.all <> status_$ok THEN RETURN;

    CASE which OF

        calc1_ec:    {when process 1 completes...}

        BEGIN

            writeln ('Processing Process 1 results');

            {get the termination status of calc1}
            pgm_$proc_wait (handle1,
                            status);

            {load the pointer array with a valid pointer}
            ec2_ptr[calc1_ec] := addr(replace_ec);

            {set the ec value to be 1 (process terminated)}
            ec2_val[calc1_ec] := 1;

            {process the results of CALC1}
                          .
                          .
                          .
        END;

       calc2_ec:  {if the process 2 completes...}

        BEGIN

            writeln ('Processing Process 2 results');

            {get the termination status of calc2}
            pgm_$proc_wait (handle2,
                            status);
```

Example 3-3. Using an Eventcount to Wait for a Child Process (Cont.)

```
            {load the pointer array with a valid pointer}
            ec2_ptr[calc2_ec] := ADDR(replace_ec);

            {set the ec value to be 1 (process terminated)}
            ec2_val[calc2_ec] := 1;

            {process the results of CALC21}
                          .
                          .
                          .

        END;
    END; {case}

    {advance the dead count}
    dead_count := dead_count + 1;
  {repeat until both processes complete }
  UNTIL (dead_count = 2) ;

END. {program}
```

Example 3-3. Using an Eventcount to Wait for a Child Process (Cont.)

3.2.3. Invoking a Program in Background Mode

To invoke a user program in background mode, call PGM_$INVOKE, with the mode option set to PGM_$BACK_GROUND. When you invoke a program this way, the invoking program creates a new process in which to run the program. Background mode differs from default mode in that a background mode process runs independently of the parent; that is, there is no communication of the completion status. If you attempt to obtain the return status of a background mode process using PGM_$PROC_WAIT, you will get an error, because the process handle is not valid for a background process.

Because a background mode process has no dependence on the parent, it is referred to as an **orphan process**. Background mode is useful for performing processing that has no further dependence on the parent process. For example, a parent process may perform interactive data collection, invoke a program in a background process to manipulate the data, then return to further data collection. This permits the data collection and data manipulation to be performed concurrently.

Example 3-4 contains two programs and a module. One program (PGM_INVOKE_DIVIDE) does the following:

- Creates an "input" file and an "error" file for use by a child process, using STREAM_$CREATE. The INVOKE_DIVIDE program will load the input file with data for the child process to use as input. The error file is for use as an error log by the child process.

- Collects data interactively -- (gets two numbers to be divided).

- Writes the data to the input file, using STREAM_$PUT_REC.

Resets the stream pointer to the beginning of the file when finished writing to the file, using STREAM_$SEEK. This is done because the stream will be passed to a child process that will read from the file. If the pointer is not RESET, the child will immediately encounter end of file.

- Invokes a program (PGM_DIVIDE) in background mode to process the information, using PGM_$INVOKE.

- Passes the background process the open stream to the input file as standard input, and passes the open stream to the error file as standard error output. (It also passes the default standard output and standard error input.)

- Continues processing.

The other program (DIVIDE) does the following:

- Establishes a fault handler to trap the divide-by-zero fault, using PFM_$ESTABLISH_FAULT_HANDLER. A fault handler must be established if you wish to log the fault before the process is terminated. The actual fault handler must be written as a separate module, and declared external. You specify the targeted fault by using the predefined fault constants in the FAULT insert file. See Chapter 2 for details about how to establish a fault handler.

- Reads the two numbers it is to divide from the standard input stream, which is the input file created and passed by INVOKE_DIVIDE.

- Divides the numbers and writes the result to standard output.

The module (PGM_ZERO_HANDLER) is the fault handler established by DIVIDE. It is invoked if the user attempts to divide by zero. It writes the fault message text to the standard error output stream, which is the error file created and passed by the parent. You must bind ZERO_HANDLER and DIVIDE befor attempting to invoke the program. See Chapter 2 for details about how to establish a fault handler.

```
  ********************
  * PGM_INVOKE_DIVIDE *
  ********************

PROGRAM pgm_invoke_divide (input, output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  status      : status_$t;

  { $CREATE variables }
  error_name  : name_$pname_t;
  error_len   : integer;
  input_name  : name_$pname_t;
  input_len   : integer;
  error_id    : stream_$id_t;
  input_id    : stream_$id_t;
  seek_key    : stream_$sk_t;
  number      : ARRAY [1..20] OF char;
  number_len  : integer32;

  { PGM_$INVOKE variables }
  handle      : pgm_$proc;   {process_handle}
  connv       : pgm_$connv;  {connection vector}
  arg_count   : pinteger;

PROCEDURE check_status; {for error handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print( status );
        pgm_$exit;
    END;
END;{check_status}

BEGIN  {main}

    { get standard error pathname for program to be invoked }
    writeln ('Input the filename to be opened as standard ',
             'error in background process DIVIDE: ');
    readln (error_name);
    error_len := SIZEOF(error_name);

    { calculate the namelength }
    WHILE ((error_name[error_len] = ' ') AND (error_len > 0)) DO
        error_len := error_len - 1;
```

Example 3-4. Invoking a Program in Background Mode

```
{ create error file - get stream }
stream_$create (error_name,
                error_len,
                stream_$write,       {access}
                stream_$unregulated, {conc}
                error_id,            {stream ID}
                status);
check_status;

{ get standard input pathname for program to be invoked }
writeln ('Input the filename to be opened as standard ',
         'input in background process DIVIDE: ');
readln (input_name);

{ calculate the namelength }
input_len := SIZEOF(input_name);
WHILE ((input_name[input_len] = ' ') AND (input_len > 0)) DO
    input_len := input_len - 1;

{create standard input file - get stream }
stream_$create (input_name,
                input_len,
                stream_$write,       {access}
                stream_$unregulated, {conc}
                input_id,            {stream ID}
                status);
check_status;

{ Get numbers to be divided by invoked program and  }
{ write them to the created standard input file.    }
writeln('input an integer to be divided:');
readln(number);

{ calculate record length }
number_len := SIZEOF(number);
WHILE ((number[number_len] = ' ') AND (number_len > 0)) DO
    number_len := number_len - 1;

{ add one for the newline }
number_len := number_len +1;
number[number_len] := CHR(10); { terminate w/ newline}

{write the number to the file}
stream_$put_rec ( input_id,      {stream to write to}
                  ADDR(number), {address of data buffer}
                  number_len,    {length of data}
                  seek_key,
                  status);
check_status;

writeln('input an integer ', number:(number_len -1),
        ' is to be divided by:');
readln(number);
```

Example 3-4. Invoking a Program in Background Mode (Cont.)

```
    { calculate record length }
    number_len := SIZEOF(number);
    WHILE ((number[number_len] = ' ') AND (number_len > 0)) DO
        number_len := number_len - 1;

    { add one for the newline }
    number_len := number_len +1;
    number[number_len] := CHR(10); { terminate w/ newline}

    {write the number to the file}
    stream_$put_rec ( input_id,
                      ADDR(number),
                      number_len,
                      seek_key,
                      status);
    check_status;

    { reset stream pointer to the beginning of the    }
    { input file before passing stream to the program }
    stream_$seek( input_id,         {stream ID}
                  stream_$rec,      {seek-base}
                  stream_$absolute, {seek-type}
                  1,                {record number}
                  status);
    check_status;

    { load $INVOKE connection vector}
    connv[0] := input_id;       { set stream IDto be created stdin  }
    connv[1] := stream_$stdout; { set stream ID to be STD_OUTPUT     }
    connv[2] := stream_$errin;  { set stream ID to be STD_ERRIN      }
    connv[3] := error_id;       { set stream ID to be created errout }

    { invoke program }
    pgm_$invoke ('pgm_divide',      { pathname of program to invoke     }
                 10,                { length of pathname                }
                 0,                 { number of arguments to be passed  }
                 0,                 { no arguments                      }
                 4,                 { number of streams to be passed    }
                 connv,             { array of stream IDS to be passed  }
                 [pgm_$back_ground],{ mode in which to invoke program   }
                 handle,            { not used in background mode       }
                 status);           { status                            }
    check_status;

    {continue processing}
            .
            .
            .
END.
```

Example 3-4. Invoking a Program in Background Mode (Cont.)

```
  **************
  * PGM_DIVIDE *
  **************
PROGRAM pgm_divide (input, output);

{Program to divide two numbers}

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/fault.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  number1 : integer;
  number2 : integer;

  status     : status_$t;
  handler_id : pfm_$fh_handle_t;

{declare external fault-handling function}
FUNCTION zero_fault_handler (IN f_status : pfm_$fault_rec_t
                            ): pfm_$fh_func_val_t; EXTERN;
BEGIN  {main}

    {establish the zero divide handler    }
    {load the target fault - 1st parameter}
    handler_id := pfm_$establish_fault_handler (fault_$zero_divide,
                                                [],          {default type}
                                                ADDR(zero_fault_handler),
                                                status);
    IF (status.all <> status_$ok) THEN
        error_$print (status);

    {read from standard input - (file passed by parent)}
    readln(number1);
    readln(number2);

    {calculate and write the result}
    write (number1:1, ' divided by ',number2:1,' is ',(number1 DIV number2):1);
    writeln (' with a remainder of ',(number1 MOD number2):1);
END.

  ****************
  * ZERO_HANDLER *
  ****************
MODULE pgm_zero_handler; {(input,output);}

{ This is a fault-handling function that prints }
{ a line and continues to fault handle.         }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pfm.ins.pas';
```

Example 3-4. Invoking a Program in Background Mode (Cont.)

```
FUNCTION zero_fault_handler (IN f_status : pfm_$fault_rec_t
                             ): pfm_$fh_func_val_t ;

BEGIN
    {write a message to the error log}
    error_$print (f_status.status);
    zero_fault_handler := pfm_$continue_fault_handling;

END; {zero_fault_handler}
```

Example 3-4. Invoking a Program in Background Mode (Cont.)

You can change a default child process into an orphan process by calling PGM_$MAKE_ORPHAN from the parent process. This option may be used for child processes that need to communicate with the parent process initially, but at some point can run independently.

PGM_$MAKE_ORPHAN takes the process handle of the child process as an input parameter. It returns a process UID that can be used to obtain information about the process (see Section 3.7). Once you convert a child process to an orphan process, the process handle is no longer valid.

The program segment in Example 3-5 demonstrates how to convert a child process into an orphan process.

```
PROGRAM pgm_orphan (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  puid     : uid_$t;
  status   : status_$t;
  handle   : pgm_$proc;

  {declare and load the standard streams}
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;    {check_status}
```

Example 3-5. Converting a Child Process to an Orphan Process

```
BEGIN {main}

   {invoke child process}
    pgm_$invoke('test5.bin', {program name}
                9,           {namelength}
                0, 0,        {no args}
                4,
                connv,       {std. streams}
                [],          {default mode}
                handle,      {process handle}
                status) ;
    check_status;

    {communicate with child}
                   .
                   .
                   .

    {cut the child loose}
    pgm_$make_orphan(handle, {process handle}
                     puid,   {process uid}
                     status);
    check_status;
                   .
                   .
                   .
END.
```

Example 3-5. Converting a Child Process to an Orphan Process (Cont.)

3.3. Passing Arguments to Invoked Programs

In addition to specifying the mode in which an invoked program is to run, PGM_$INVOKE permits the passing of arguments to the invoked program. The third and fourth parameters of the PGM_$INVOKE call are the argument count and argument vector, respectively. The argument count is a 2-byte integer specifying the number of arguments being passed. The argument vector is an array of pointers to the arguments being passed. The argument vector is of the type PGM_$ARGV, which is an array of UNIV_PTR types.

A program can pass any number of arguments to a program it is invoking. However, when passing arguments to a Shell, the Shell's syntax limits the number of arguments to 10 (including program name). Each argument must be preceded by a 2-byte integer indicating the number of bytes in the argument. The first argument must be the name of the program -- the simple name, not the full pathname (i.e., date, not //deedle/com/date).

DOMAIN provides a predefined record type, PGM_$ARG, which is a 128-byte character array preceded by a 2-byte integer. Whether you choose to use the predefined argument type, or declare a argument type of your own, will depend on the length of the passed arguments and how critical storage is to your program.

Figure 3-1 illustrates the argument vector/argument arrangement.

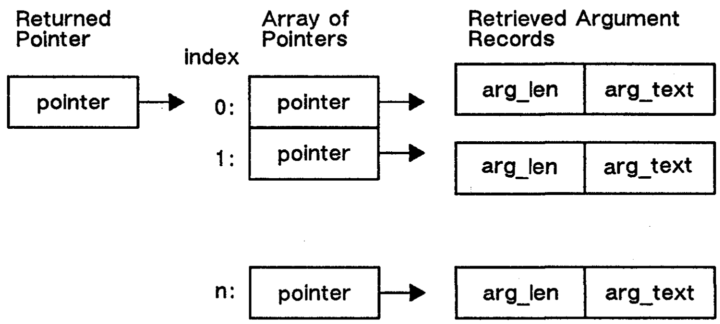


Figure 3-1. Argument Vector/Argument Configuration

The program in Example 3-6 invokes a program (in a child process) and passes two arguments: the invoked program name and a text string. (Remember, the name of the invoked program must be passed as the first argument.)

```
PROGRAM pgm_pass_args (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  status : status_$t;

  {argument variables}
  name,
  argument : pgm_$arg;

  {INVOKE variables}
  argv   : pgm_$argv;
  handle : pgm_$proc;

  {declare and load the standard streams}
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];


PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;
```

Example 3-6. Passing Arguments to an Invoked Program

```
BEGIN     {main program}

    {load the arguments}
    name.chars := 'pgm_passee.bin';
    name.len   := 14;
    argument.chars := 'test';
    argument.len   := 4;

    {load the argument vector w/ addresses}
    argv[0] := ADDR(name);
    argv[1] := ADDR(argument);

    pgm_$invoke('pgm_passee.bin', {process name}
                 14     ,          {name length}
                 2,                {arg count - name & arg}
                 argv,             {arg vector}
                 4,                {stream count}
                 connv,            {std. streams}
                 [],               {mode}
                 handle,           {process handle}
                 status) ;
    check_status;
                .
                .
                .
    pgm_$proc_wait (handle,   {process handle}
                    status);
    check_status;

END.
```

Example 3-6. Passing Arguments to an Invoked Program (Cont.)

3.4. Accessing Arguments from an Invoked Program

An invoked program can access the arguments passed to it in two ways:

- Calling PGM_$GET_ARG, which returns one argument at a time.
- Calling PGM_$GET_ARGS, which returns a pointer to an array containing all the passed arguments.

3.4.1. Accessing Arguments with PGM_$GET_ARG

PGM_$GET_ARG is a function that returns an argument and its length. To access an argument with it, specify the argument vector index number of the pointer to the argument, and the maximum length of the argument. For example, to index the program name, which is the first argument, specify the index number as 0 and a maximum length that will accommodate the name.

Example 3-7 shows a program that could be invoked by a program similar to the one in Example 3.3. This program accesses the second argument in the argument array. (Typically, the program name is ignored by an invoked program.)

```
PROGRAM pgm_passee_arg (input, output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  status     : status_$t;
  arg_length : pinteger;              {returned argument length}
  arg_num    : pinteger;              {ordinal # of desired argument}
  argument   : array [1..256] of char; {argument buffer}
  max_len    : pinteger := 256;       {maximum length of returned arg}

BEGIN
          .
          .
          .
    {access 2nd argument}
    arg_num := 1;   {2nd arg #, 0 is 1st}

    arg_length := pgm_$get_arg (arg_num,  {arg number}
                                argument, {arg buffer}
                                status,
                                max_len);

    writeln ('this is the second argument: ', argument:arg_length);

    IF status.all <> status_$ok THEN
        error_$print (status);
    {process the argument}
          .
          .
          .
END.
```

Example 3-7. Accessing Arguments with PGM_$GET_ARG

3.4.2. Accessing Arguments with PGM_$GET_ARGS

PGM_$GET_ARGS returns a pointer to the argument vector, and the number of pointers in the vector (the number of arguments passed).

The program in Example 3-8 may also be invoked by a program similar to the one in Example 3.3. It accesses both arguments passed to it.

Note that the argument vector is a PGM_$ARGV data type. This is an array of addresses in UNIV_PTR format. You cannot dereference a UNIV_PTR. So, to access the argument you must:

1. Declare an explicit type pointer for the arguments.

2. Typecast the UNIV_PTRs to be explicit pointers.

3. Dereference the explicit pointers.

The program segment in Example 3-8 accesses arguments with PGM_$GET_ARGS, and writes them to output.

```
PROGRAM pgm_passee (input, output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

TYPE
  {declare an explicit argument pointer}
  pgm_arg_ptr = ^pgm_$arg;

VAR

  arg_count    : pinteger;      {argument count}
  arg_vec_addr : pgm_$argv_ptr; {argument vector}
  i            : pinteger;      {index}

  {declare array to hold arguments}
  arguments    : array [0..127] of pgm_arg_ptr;

BEGIN

    {get a pointer to the argument array}
    pgm_$get_args (arg_count,     {number of arguments}
                   arg_vec_addr); {returned pointer}

    FOR i := 0 TO (arg_count - 1) DO BEGIN

        {typecast the pointer and load into argument array}
        arguments[i] := pgm_arg_ptr( arg_vec_addr^[i]);

        {write argument to output (dereference explicit pointer)}
        writeln ('Argument ', i:1,' is ', arguments[I]^.chars:arguments[I]^.len);
    END;
                .
                .
                .
END.
```

Example 3-8. Accessing Arguments with PGM_$GET_ARGS

3.5. Deleting Arguments

DOMAIN provides the call PGM_$DEL_ARG to delete arguments from the argument vector. PGM_$DEL_ARG is useful in the case of invoking a program (for example, PROG_A) that invokes another program (PROG_B). In this instance, you can pass PROG_A the arguments needed for both programs. PROG_A uses PGM_$DEL_ARG to delete the arguments it uses from the argument vector, then uses the modified vector to invoke PROG_B.

The DOMAIN Language Level Debugger (DEBUG) is an example of such a program. Consider the following Shell command:

debug -src taxes.bin income

This command invokes the debugger with an argument vector that contains pointers to all four elements of the command. All four elements are arguments to the DEBUG program. However, before invoking the user program taxes.bin, the debugger deletes "debug" and "-src" from the argument vector.

To delete an argument from the argument vector, call PGM_$DEL_ARG specifying the index number of the argument pointer in the argument vector. For example, to delete the first argument, specify 0 as the index number.

The program in Example 3-9 is passed an argument vector that contains two arguments, its name and the name of a program it invokes. The example accesses the argument vector using PGM_$GET_ARGS, deletes the name argument, then invokes the other program, using the same argument vector. In a more complex program, you might read each argument, searching for a flag that separates the arguments of the two programs.

```
PROGRAM pgm_del_inv (input, output);
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

TYPE
  {construct a pointer to arguments}
  pgm_arg_ptr = ^pgm_$arg;

VAR
  arg_count    : pinteger;
  arg_vec_addr : pgm_$argv_ptr;

  {declare array to hold arguments}
  i            : integer;
  arguments    : array [0..127] of pgm_arg_ptr;

  {INVOKE variables}
  status : status_$t;
  handle : pgm_$proc;
  {declare and load the standard streams}
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;

BEGIN {main}

    writeln ('In del_inv');

    pgm_$get_args (arg_count,     {number of arguments}
                   arg_vec_addr); {pointer to argument vector}

    writeln('passed folowing arguments:');
```

Example 3-9. Deleting an Argument from the Argument Vector

```
    FOR i := 0 TO (arg_count - 1) DO BEGIN
        arguments[i] := pgm_arg_ptr( arg_vec_addr^[i]);
        writeln('ARG ', i:1, ' ', arguments[i]^.chars : arguments[i]^.len);
    END;

    {delete program name argument}
    writeln;
    writeln('deleting ARG 0');
    pgm_$del_arg (0);

    {GET_ARGS passes UNIV pointer to the argument array. To      }
    {reference arguments, you must typecast to pgm_$arg pointers }

    FOR i := 0 TO (arg_count - 1) DO BEGIN
        arguments[i] := pgm_arg_ptr( arg_vec_addr^[i]);
    END;

    writeln('invoking ',arguments[0]^.chars:arguments[0]^.len,' (now arg 0)');
    writeln;

    {invoke second program w/ modified arg vector}

    pgm_$invoke( arguments[0]^.chars, {process name}
                 arguments[0]^.len,   {name length}
                 1,                   {arg count - name}
                 arg_vec_addr^,       {arg vector}
                 4,                   {stream count}
                 connv,               {std streams}
                 [pgm_$wait],         {mode}
                 handle,              {process handle}
                 status) ;
    check_status;
                 .
                 .
                 .
END.
```

Example 3-9. Deleting an Argument from the Argument Vector (Cont.)

3.6. Passing Streams to an Invoked Program

PGM_$INVOKE also permits the passing of streams to the invoked program. The fifth and sixth parameters of the INVOKE call are the stream count and connection vector, respectively. The stream count is a 2-byte integer specifying the number of streams being passed. The connection vector is an array of stream IDS, in PGM_$CONNV format. Stream IDS refer to objects already opened by the calling program, using STREAM_$CREATE or STREAM_$OPEN. The first element in the connection-vector array becomes stream 0 in the invoked program, the second element becomes stream 1, and so on.

By default, every program is invoked with four streams, numbered 0 through 3. Stream 0 is standard input, stream 1 is standard output, stream 2 is error input, stream 3 is error output. To invoke a program with these four streams, pass the predefined standard stream constants.

You may also leave "holes" in the connection vector, by setting a stream ID equal to the predefined constant, STREAM_$NO_STREAM. (The STREAMS insert file must be included to use this constant.)

The program in Example 3-10 opens a file and passes the stream ID of the file as standard output. Note that the STREAM_$NO_STREAM constant is used to pass a null stream as the standard input.

```
PROGRAM pgm_pass_streams (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  status : status_$t;
  {argument variables}
  name     : pgm_$arg;
  argument : pgm_$arg;

  {INVOKE variables}
  argv   : pgm_$argv;
  connv  : pgm_$connv;
  handle : pgm_$proc;

  {CREATE variables}
  pathname   : name_$pname_t;
  namelength : integer;
  stream_id  : stream_$id_t;

PROCEDURE check_status;  {for error_handling}
BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;

BEGIN     {main program}

    {get the input}
    writeln ('Enter the output file pathname: ');
    readln (pathname);

    { calculate the length of pathname }
    namelength := SIZEOF(pathname);
    WHILE (pathname[namelength] = ' ') AND (namelength > 0) DO
        namelength := namelength - 1;

    {open w/ $CREATE}
    stream_$create (pathname,
                    namelength,
                    stream_$write,              {access}
                    stream_$controlled_sharing, {conc}
                    stream_id,
                    status);
    check_status;
```

Example 3-10. Passing Streams to an Invoked Process

```
    {load the arguments}
    name.chars := 'pgm_passee.bin';
    name.len   := 14;
    argument.chars := 'test';
    argument.len   := 4;

    {load the argument vector w/ addresses}
    argv[0] := ADDR(name);
    argv[1] := ADDR(argument);

    {load connection vector}
    connv[0] := stream_$no_stream; {null stream}
    connv[1] := stream_id;         {pass stream ID as stdout}
    connv[2] := stream_$errin;
    connv[3] := stream_$errout;

    pgm_$invoke('pgm_passee.bin', {process name}
                14,               {name length}
                2,                {arg count - name & arg}
                argv,             {arg vector}
                4,                {stream count}
                connv,            {connection vector}
                [],               {mode}
                handle,           {process handle}
                status) ;
    check_status;

    {get process termination status}
    pgm_$proc_wait (handle,   {process handle}
                    status);
    check_status;
                 .
                 .
                 .
END.
```

Example 3-10. Passing Streams to an Invoked Process (Cont.)

3.7. Getting Process Information

You can obtain information about your process and other processes on your node by using calls from the PGM, PM, PROC1, and PROC2 subsystems.

3.7.1. Getting Information About Your Process

The following calls return information about the process that calls them:

PM_$GET_HOME_TXT Returns the home directory as a string.

PM_$GET_SID_TEXT Returns the SID (login identifier) as a string.

PROC1_$GET_CPUT Returns the CPU time used by the process.

PROC2_$GET_INFO Returns a record containing the following information:

- The program state (ready, waiting, suspended, susp_pending, bound).
- The User Status Register (USR).
- The User Program Counter (UPC).
- The user stack pointer (A7).
- The stack base pointer (A6).
- The amount of CPU time used.
- The CPU scheduling priority.

To obtain either the home directory or SID, call PM_$GET_HOME_TEXT or PM_$GET_SID_TEXT, respectively, specifying a maximum length for the string buffer to hold the returned data. The calls return the requested string along with the actual length of the string.

To obtain the CPU time used by your process, call PROC1_$GET_CPUT, specifying an output parameter in TIME_$CLOCK_T format.

To obtain the information record for your process, you must pass PROC2_$GET_INFO the UID of your process and the buffer length for the record. Your process UID is obtained by calling PROC_$WHO_AM_I, which has one parameter -- the returned process UID. Specify a length of 36 bytes for the information record buffer.

The program in Example 3-11 gets the home directory text, the process SID, the total CPU time, and the information record, and prints the information to standard output.

```
PROGRAM pgm_your_proc (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/proc1.ins.pas';
%include '/sys/ins/proc2.ins.pas';
%include '/sys/ins/pm.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/type_uids.ins.pas';

VAR
  home       : string;
  home_len   : pinteger;
  sid        : string;
  sid_len    : pinteger;
  uid        : uid_$t;
  info       : proc2_$info_t;
  status     : status_$t;
  total_time : time_$clock_t;
  d_clock    : cal_$timedate_rec_t;

BEGIN

    pm_$get_home_txt (30,       {maxlen}
                      home,     {dir}
                      home_len);

    writeln ('home directory ', home : home_len);

    pm_$get_sid_txt (40,    {maxlen}
                     sid,   {dir}
                     sid_len);


    writeln ('sid ', sid : sid_len);

    proc2_$who_am_i (uid);

    writeln ('uid ', uid.high, uid.low);

    proc2_$get_info (uid,       {process uid}
                     info,
                     36,        {info buffer length}
                     status);

    IF (status.all <> proc2_$is_current) THEN
        error_$print (status);

    {write the information}
    writeln ('stack uid ', info.stack_uid.high);
    writeln ('stack uid ', info.stack_uid.low);
    writeln ('stack base ', info.stack_base);
```

Example 3-11. Getting Information About Your Process

```
    IF proc2_$waiting IN info.state THEN
        writeln ('state: waiting');
    IF proc2_$suspended IN info.state THEN
        writeln ('state: suspended');
    IF proc2_$susp_pending IN info.state THEN
        writeln ('state: susp_pending');
    IF proc2_$bound IN info.state THEN
        writeln ('state: bound');

    writeln ('user sr ', info.usr);
    writeln ('user pc ', info.upc);
    writeln ('user stack pointer ', info.usp);
    writeln ('sb ptr ', info.usb);

    {decode the time}
    cal_$decode_time (info.cpu_total,
                      d_clock);

    writeln ('cum cpu: ',d_clock.hour:1,' ',
                         d_clock.minute:1,' ',
                         d_clock.second:1,' ');
    writeln ('priority ', info.priority:1);
    writeln ;

    proc1_$get_cput (total_time);

    {decode the time}
    cal_$decode_time (total_time,
                      d_clock);

    writeln ('GET_CPU total time: ',d_clock.hour:1,' ',
                                    d_clock.minute:1,' ',
                                    d_clock.second:1,' ');

END.
```

Example 3-11. Getting Information About Your Process (Cont.)

3.7.2. Getting Information About Other Processes

You can also obtain process information about:

- Processes invoked by your process.
- All other user processes on the same node as your process.

To obtain process information about a process invoked by your process:

1. Call PGM_$GET_PUID specifying the process handle of the child process as an input parameter. (The process handle is returned when you invoke a process using PGM_$INVOKE.) PGM_$GET_PUID returns the UID of the specified process.
2. Call PROC2_$GET_INFO, using the returned UID.

To obtain information about all user processes running on the same node as your process:

1. Call PROC2_$LIST, specifying a maximum number of UIDS you want returned. PROC2_$LIST returns the UIDS of all the user processes running on the same node as the calling process, in an array of PROC2_$UID_LIST_T format.

2. Call PROC_$GET_INFO once for each returned UID.

The program in Example 3-12 invokes a program in a child process, gets the information record of the invoked process, and writes the accumulated CPU time of the process (a field in the information record) to standard output.

```
PROGRAM pgm_child_info (input,output);

{ This program gets the amount of time the child has used}

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/proc2.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  status     : status_$t;
  proc_uid   : uid_$t;              {process uid}
  info       : proc2_$info_t;       {information record}
  total_time : time_$clock_t;       {encoded time}
  d_clock    : cal_$timedate_rec_t; {decoded time}
  rel_time   : time_$clock_t;       {relative amount of time}
  handle     : pgm_$proc;           {process handle}

  {declare and load the standard streams}
  connv  : pgm_$connv :=
           [stream_$stdin, stream_$stdout,
            stream_$errin, stream_$errout];

PROCEDURE check_status;  {for error_handling}

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print (status);
        pgm_$exit;
    END;
END;
```

Example 3-12. Getting Information About an Invoked Process

```
BEGIN
    pgm_$invoke('calc.bin', {process name}
                8,           {name length}
                0,0,         {no args}
                4,           {stream count}
                connv,       {std streams}
                [],          {default mode}
                handle,      {process handle}
                status) ;
    check_status;

    {wait 10 seconds}
    {convert # of seconds to UTC value}
    cal_$sec_to_clock (10,
                       rel_time);

    time_$wait (time_$relative,     {pre-defined}
                rel_time,           {time to wait}
                status);

    {perform other processing}
                .
                .
                .

    {get the process uid}
    pgm_$get_puid (handle,    {process handle}
                   proc_uid,  {process uid}
                   status);
    check_status;

    {get process information}
    proc2_$get_info (proc_uid, {process uid}
                     info,
                     36,       {info buffer length}
                     status);
    check_status;

    {decode the cpu time}
    cal_$decode_time (info.cpu_total,
                      d_clock);

   vfmt_$write5 ('Accumulated CPU time of Child : %2ZWD:%2ZWD:%2ZWD %.',
                 d_clock.hour,
                 d_clock.minute,
                 d_clock.second,
                 0,0);             {dummy arguments}

    {get child's terminaton status}
    pgm_$proc_wait (handle, {process handle}
                    status);
    check_status;
                .
                .
                .
END.
```

Example 3-12. Getting Information About an Invoked Process (Cont.)

Chapter 4
Performing I/O with IOS Calls

The IOS interface consists of IOS system calls that allow you to create, read, write, and delete objects by opening stream connections to them. A **stream connection** is a pathway from the program that is manipulating the object to the disk file or I/O device where the object is physically located. You can read and change the attributes of an object and its stream connection. This allows you to control what operations can be performed on an object, and how your program and other programs can access it.

Usually, you can perform I/O operations using statements and functions in your high-level language. And, in fact, you want to use high-level langauge I/O if you are most concerned about transporting your programs to other operating systems.

However, DOMAIN provides this IOS interface to perform I/O operations if your high-level language does not provide a way, is less convenient to use, or if using it would introduce undesirable peculiarities on certain devices.

IOS calls can sometimes be more efficient than language I/O. For example, the IOS manager provides a call that allows you to read data without having to copy the data into a buffer. Standard UNIX I/O does not provide a comparable feature.

This chapter describes the most common calls in the IOS interface. It describes how to create, open, close, read, write, and delete various types of objects using IOS calls.

4.1. System Calls, Insert Files, and Data Types

To perform system I/O, use system calls with the prefix IOS. In order to use IOS system calls, you must include the appropriate insert file in your program. The IOS insert files are:

/SYS/INS/IOS.INS.C	for C programs.
/SYS/INS/IOS.INS.FTN	for FORTRAN programs.
/SYS/INS/IOS.INS.PAS	for Pascal programs.

Note that some IOS system calls require that you specify a type UID. To use standard DOMAIN types, you must include the appropriate type UID insert file for your program:

/SYS/INS/TYPE_UIDS.INS.C	for C programs.
/SYS/INS/TYPE_UIDS.INS.FTN	for FORTRAN programs.
/SYS/INS/TYPE_UIDS.INS.PAS	for Pascal programs.

This chapter is intended to be a guide for performing certain programming tasks; the data type and system call descriptions in it are not comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference* manual.

4.2. Overview of the IOS Manager

The IOS interface is actually part of a larger facility that DOMAIN provides to perform stream I/O. The **Streams** facility allows DOMAIN programs to perform I/O on various types of objects. Among the object types that DOMAIN defines is the unstructured ASCII (UASC) type, serial I/O line (SIO) type, and the record (REC) type. (See Section 4.3 for a more complete list.)

The Streams facility is designed so that it can insulate the I/O operation from the type of object it is operating on. For example, a program can use the same I/O statement to write to an object, regardless of whether the object's type is UASC or MBX. Whenever a program performs an I/O operation, the Streams facility recognizes the object type being manipulated and calls a corresponding **type manager**. The type managers define how the I/O operations can be performed on that particular object type. The managers actually perform the I/O operation by making calls to more primitive (or device-dependent) managers. For example, the UASC type manager uses MS calls to perform an I/O operation on a UASC object while the MBX type manager uses MBX calls to perform an I/O operation on an MBX object. This layered approach allows application programmers to use various object types without having to know the details of how I/O for each type is implemented.

Another advantage of having the Streams facility comprised of various type managers is that users, as well as DOMAIN, can define new object types and write new type managers as the need arises. For information on writing a type manager see the *Using the Open System Toolkit to Extend the Streams Facility* manual.

Generally, when using IOS calls, you need not be concerned about the other parts of the Streams facility. The Streams facility does the work for you. Whenever a program performs an I/O operation, (either by using a language I/O statement such as Pascal's *writeln*, DOMAIN/IX's *write*, or by an IOS call such as IOS_$PUT) the Streams facility recognizes the object type being manipulated and calls the appropriate type manager for that type.

You can use IOS calls as a way of making your program *generic* or less dependent on any specific device, or manager. You can do so because most of the IOS calls perform the same way, regardless of the object type you are using. This chapter describes the basic IOS calls independent of any objects. Chapter 9 describes how to use IOS calls to access the object types that DOMAIN supports.

Before we describe how to use IOS calls to perform system I/O, we must first define a few terms. The following sections define some of the basic features of the IOS interface:

- Stream connections
- Stream IDs
- Default Stream IDs
- Stream markers

4.2.1. Stream Connections

A **stream connection**, often referred to as simply a **stream**, is a pathway to an object such as a disk file or I/O device. This is how your program *connects* to the object. Whenever a program wants to perform I/O on an object, the program must first make one or more stream connections to that object. You establish a stream connection when you open the object using IOS_$CREATE or IOS_$OPEN.

4.2.2. Stream IDs

You make a connection when you create or open an object, specifying the pathname of the desired object. If the call succeeds, it returns an identification number or **stream ID**. The stream ID identifies the stream connection to the calling program. You use the returned stream ID as an input parameter to any system calls requiring a stream ID. (IOS, SIO, PAD, and some GPR system calls require that you specify a stream ID.)

Once a program makes a stream connection, the program uses the stream ID, not the pathname, to perform I/O on the associated object. The program terminates the stream connection when it performs a *close* operation (for IOS calls, IOS_$CLOSE closes the specified stream connection).

Note that stream IDs are not the same as FORTRAN logical unit numbers, which are channel numbers that the *programmer* selects. In contrast, stream IDs are assigned by the Streams facility.

4.2.3. Default Stream IDs

Typically, a program's runtime environment requires a specific set of stream connections, so the IOS manager provides these by default. Each time you create a process, IOS opens these default streams for program input and output:

- Standard input
- Standard output
- Error input
- Error output

Standard input and **standard output** are streams that channel normal input and output between a user and a process. By default, standard input is an input pad. Standard output is a transcript pad.

Shell commands use input and output streams when processing command line data. When a user specifies a command in the Shell input pad, standard input passes data from the command line to the command program. Standard output passes data from the program to the transcript pad.

Error input and **error output** are streams that handle additional program input and output. By default, error input is an input pad. Error output is a transcript pad.

An error input stream has nothing to do with errors; it is simply an additional input stream to pass data to a program. For example, when a command queries a user to verify wildcard names, error input passes the user's response to the command program. Error output is the stream that passes program error messages to the process transcript pad.

Table 4-1 lists the default streams by their predefined names, and actual stream number. These constants are defined in the BASE insert files for each programming language.

Table 4-1. Default Streams

Stream	IOS Defined Value	Number
Standard input	IOS_$STDIN	(0)
Standard output	IOS_$STDOUT	(1)
Error input	IOS_$ERRIN	(2)
Error output	IOS_$ERROUT	(3)

In some cases, you may want to redirect standard input and output to read input from and write data to locations other than the process input and transcript pads. For example, your program might expect data from a disk file rather than from a user at the keyboard. The Shell allows users to redirect standard input and output with the I/O control characters such as < and >.

You can redirect standard input and standard output stream connections by assigning a different stream ID to the stream connection. You can also redirect any standard stream using PGM_$INVOKE. For details, see Chapter 3 of this manual. A single process can have a maximum of 127 stream IDs open at one time.

Note that when you redirect standard input or standard output, the error input and error output keep their original connection. Some programs use error input and error output as *interactive* connections, and standard input and standard output for the remaining data I/O. For example, if a user has redirected standard input to a disk file, the program uses error input to get information from the user (the keyboard) rather than from the file.

4.2.4. Stream Markers

Every open stream has a **stream marker** that points to the current position in an object. When you open a stream to an object, the stream marker usually starts at the beginning of the object (BOF). However, if your program wants to add data at the end of an existing object, you can specify that the stream marker's initial position be at the end of the object (EOF).

The stream marker moves as you perform read or write operations on the object. When you read from the object, the stream marker always moves so that it points to the data item you would read next. The IOS manager returns an error if you try to read data when a stream marker is pointing to EOF.

Many stream operations refer to the stream marker to complete the operation. Your programs can inquire about, and explicitly move the stream marker, by using the IOS_$SEEK calls. (For details, see Section 4.9).

For some types of objects, like UASC objects, the stream marker keeps track of the current stream position. For other types of objects, like an SIO line, the stream marker is irrelevant.

4.2.5. IOS Calls for Manipulating Streams

The IOS manager provides a few calls that allow you to to manipulate stream IDs or make copies of stream connections. Table 4-2 lists the calls you can use.

Table 4-2. IOS Calls to Manipulate Stream Connections

IOS Call	Description
IOS_$EQUAL	Determines whether two stream IDs refer to the same object. (Useful to avoid using two streams when one is sufficient.)
IOS_$SWITCH	Switches a stream connection from one stream ID to another stream ID. The new stream ID refers to the same connection as the old stream ID, making the old stream ID invalid.
IOS_$DUP	Creates a copy of a specified existing stream ID. The new stream ID refers to the same connection as the existing stream ID.
IOS_$REPLICATE	Creates a copy of a specified existing stream ID. The new stream ID refers to the same connection as the existing stream ID.

Note that IOS_$DUP is identical to IOS_$REPLICATE except that IOS_$DUP looks for a free stream number in *ascending* order from the specified stream ID, while IOS_$REPLICATE looks in *descending* order. IOS_$DUP is analogous to UNIX's DUP function.

You use either IOS_$DUP or IOS_$REPLICATE to *copy* existing stream IDs -- both the existing and new stream IDs remain valid connections. Typically, you copy a stream to keep the connection open when passing it to a subroutine. By copying the stream before passing it, you prevent the subroutine from closing your connection to the object. Even if the subroutine closes its connection, you will still have a valid stream ID for an open stream.

You use IOS_$SWITCH to *replace* stream IDs; you *switch* the connection from the existing stream ID to the new stream ID.

4.3. Creating and Opening Objects

The IOS manager provides two calls to open objects:

IOS_$CREATE Creates an object if it does not exist, or opens an existing object.

IOS_$OPEN Opens an object only if it exists. The call returns an error if the object you specify does not exist.

IOS_$CREATE allows you to create an object of any type defined by a user or DOMAIN (for example, UASC, record, or MBX objects). An object's type determines how IOS calls work for that object. For example, IOS calls can support seek operations if you create a UASC object, but not if you create an MBX object.

You can specify various actions to take if your program tries to create an object with a name that already refers to an existing object. For example, you can create temporary or backup versions of existing objects. You control how IOS_$CREATE opens existing objects by specifying appropriate create modes.

When opening the object using either IOS_$CREATE or IOS_$OPEN, you can control certain aspects of the open stream connection. For example, you can specify how your program can access the object and whether other programs can access the object at the same time. You control how to open an object by specifying the appropriate open options.

The following sections describe the create and open calls in detail:

- Section 4.3.1 describes how to create an object of a particular type with IOS_$CREATE.

- Section 4.3.2 describes how to use the create modes to control how IOS_$CREATE opens an object if it already exists.

- Section 4.3.6 describes the open options that you can specify with either IOS_$CREATE and IOS_$OPEN.

4.3.1. Specifying an Object's Type

The IOS manager allows you to operate on many types of objects. As an application programmer, you will see that most of the IOS calls work the same way regardless of the object type you are using (unless the type manager does not support the IOS operation). This allows you to design your program independent of any implementation details specific to a particular object type.

To handle the specifics of each type, the IOS manager directs each IOS call to the appropriate type manager for that type. The type manager actually performs the I/O operation according to its implementation. For example, when your program uses IOS_$CREATE to create a UASC object, the IOS manager directs the call to the UASC type manager. The UASC type manager creates the UASC object by making subsequent MS calls. In contrast, if the program uses IOS_$CREATE to create a mailbox object, the IOS manager directs the call to the MBX type manager.

The IOS manager recognizes the object's type by checking its **type UID**. A type UID is a number that uniquely identifies a class of objects. You can specify the type of object that you want to operate on when you create the object. You supply the object type, in UID_$T format, of a system object in the third parameter of the create call. Table 4-3 lists some of the object types defined by DOMAIN with their predefined constants. Chapter 9 describes the types of objects defined by DOMAIN in detail.

Note that the following is only a partial list of type UIDs because users, as well as DOMAIN, can add a new object type whenever the need arises by writing a type manager. DOMAIN provides the Open System Toolkit to help you define your own I/O operations. See the *Using the Open System Toolkit to Extend the Streams Facility* manual for details.

When using any IOS calls that require you to specify a type UID, you might need to include a type UID insert file. The standard DOMAIN types are defined in the TYPE_UIDS.INS.xxx insert file, where xxx stands for the language extension,. C, .FTN, or .PAS.

Currently, the only IOS call that requires you to specify a type UID is IOS_$CREATE. Even then, you don't have to specify the type UID insert file in programs that use IOS_$CREATE when you create an object of the default type. You specify the default type, which is currently the UASC object type, by specifying the predefined value, UID_$NIL. UID_$NIL is declared in the BASE insert file.

Most of the examples in this chapter manipulate this default type. See Chapter 9 for information on using IOS calls to access other types of objects such as mailboxes, serial lines, and magnetic tapes.

Table 4-3. Object Types

Type UID	Object
UASC_$UID	UASC object
RECORDS_$UID	Record-structured object
HDR_UNDEF_$UID	Nonrecord-structured object
OBJECT_FILE_$UID	Object module object (compiler or binder output)
SIO_$UID	Serial line descriptor object
MT_$UID	Magnetic tape descriptor object
PAD_$UID	Saved Display Manager transcript pad
INPUT_PAD_$UID	Display Manager input pad
MBX_$UID	Mailbox object
DIRECTORY_$UID	Directory
NULL_$UID	Null device

4.3.2. Controlling how IOS Creates Objects

You can specify various actions to take if your program tries to create an object with a name that already refers to an existing object. For example, a user of your program might specify a name of an object not knowing it already exists. Your program can either create a new version of that object, open a stream to the existing object, or return an error indicating that the object already exists.

You control how IOS_$CREATE creates an object by specifying one or more of the create modes in the fourth parameter of your call. Table 4-4 lists the modes, in IOS_$CREATE_MODE_T format, that control how IOS_$CREATE creates new objects if the name specified refers to an object that already exists. If a name does not refer to an existing object, IOS_$CREATE just creates a new object, ignoring any create modes.

Table 4-4. Controlling IOS_$CREATE when a Name Refers to an Existing Object

IOS_$NO_PRE_EXIST_MODE	Returns the IOS_$ALREADY_EXISTS error status code, if an object with the specified name already exists.
IOS_$PRESERVE_MODE	Preserves the contents of the object, if an object with the specified name already exists. It then opens the object and positions the stream marker to the begining of the object (BOF) unless you set the IOS_$POSITION_TO_EOF open option. Use this mode to change or add data to an existing object. (See Section 4.3.6 for details on open options.)
IOS_$RECREATE_MODE	Recreates the object if an object with the specified name already exists. Essentially, this option deletes the existing object and creates a new one. Use this mode to create the object as if the name never existed. The object created will have the default set of attributes for that object type.
IOS_$TRUNCATE_MODE	Opens the object and deletes the contents, if an object with the specified name already exists. Use this mode to create an object that has the same attributes as the object with the specified name.
IOS_$MAKE_BACKUP_MODE	Creates a temporary object with the same type and attributes as the object specified in the pathname, if an object with the specified name already exists. Use this mode to create a backup object.

Section 4.3.3 describes how to create a backup version of an existing object in detail. Section 4.3.4 describes how to use an additional create mode, IOS_$LOC_NAME_ONLY_MODE, which determines how IOS_$CREATE creates a temporary object.

4.3.3. Creating a Backup Object

To create a backup version of a specified object, use IOS_$CREATE with the IOS_$MAKE_BACKUP_MODE create mode. The new object is the same as the object specified by "pathname" (if it exists) in that it has the same type and other attributes, and it is created on the same volume (node).

IOS_$CREATE (with IOS_$MAKE_BACKUP_MODE) does not open or modify the object specified by the pathname, but it examines the object to extract its attributes. Even though the call doesn't modify the object, it conceptually replaces the object, so this operation requires write access to object.

When you close this stream with an IOS_$CLOSE, IOS_$CLOSE changes the object specified by "pathname" to "pathname.bak." It changes the new (formerly the temporary, unnamed) object to "pathname," and makes the object permanent. If a ".bak" version of the object already exists, IOS_$CLOSE deletes it. (The caller must have either D or P rights to delete the object.) If the ".bak" object is locked at the time IOS_$CLOSE is called, the object will be deleted when it is unlocked.

If the object doesn't exist, IOS_$CREATE creates the object specified by "pathname," and IOS_$MAKE_BACKUP_MODE has no effect.

4.3.4. Creating Temporary Objects

IOS_$CREATE allows you to create a temporary object two ways. To create a temporary object on your *boot volume*, specify a null value for a pathname and a value of 0 in namelength. To create a temporary object on *another volume*, specify the pathname of an existing object on that volume and the IOS_$LOC_NAME_ONLY_MODE create mode. IOS_$CREATE creates a temporary unnamed object on the same node as the object you specify in "pathname."

4.3.5. Examples of Opening and Creating Objects

Example 4-1 is a program segment that calls IOS_$CREATE to create a UASC object, or open one if it already exists. The program calls IOS_$CREATE with the IOS_$PRESERVE_MODE create mode to save the contents of the object (if it exists) and the IOS_$POSITION_TO_EOF_OPT open option to position the stream marker at the end of the object. This causes IOS_$PUT to append data to the end of the object. Since we use IOS_$CREATE, the object is automatically open for write access. See Section 4.3.7 for more information about controlling an object's read and write access.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR

  status : status_$t;
  count  : integer;

  {$CREATE variables}

  pathname     : name_$pname_t;
  namelength   : integer;
  type_uid     : uid_$t;
  stream_id    : ios_$id_t;

PROCEDURE check_status;  { for error handling }
 .
 .
 .
```

Example 4-1. Creating an Object

```
BEGIN    {main}

    { Get the pathname. }
    writeln;
    writeln ('Type the pathname of object to create or open: ');
    namelength := SIZEOF(pathname);

    { Convert pathname to internal format using VFMT_$READ. }
    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);

    { Create the object, or open an existing object for appending input. }
    ios_$create (pathname,
                 namelength,
                 uid_$nil,             { Default type UID (UASC) }
                 ios_$preserve_mode,        { Open object if exists }
                 [ios_$position_to_eof_opt], { Append data at end }
                 stream_id,
                 status);
    check_status;
.
.
.
```

Example 4-1. Creating an Object (Concluded)

The program segment in Example 4-2 asks the user to specify an existing object. It then opens the object using IOS_$OPEN with write access and sets the stream marker to EOF to append data. If it opened the object for write access without specifying IOS_$POSITION_TO_EOF_OPT, the data would be overwritten. The next section describes the IOS_$OPEN call in detail. See Section 4.3.7 for more information about controlling a stream's read and write access.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  status : status_$t;
  count  : integer;

  {$OPEN variables}
  pathname     : name_$pname_t;
  namelength   : integer;
  open_opt     : ios_$open_options_t;
  stream_id    : ios_$id_t;

BEGIN    {main}

    {Get the pathname. }
    writeln ('Type the name of the existing object you want to open: ');
```

Example 4-2. Opening an Existing Object

```
{ Convert pathname to internal format using VFMT_$READ. }
namelength := sizeof(pathname);
vfmt_$read2('%""%eka%.',
            count,
            status,
            pathname,
            namelength);

{ Open the object.}
stream_id := IOS_$OPEN ( pathname,
                         namelength,
                         [ios_$write_opt,          { Open with write access }
                          ios_$position_to_eof_opt], { Append data at end }
                         status);
check_status;
    .
    .
    .
```

Example 4-2. Opening an Existing Object (Concluded)

4.3.6. Controlling how IOS Opens Objects

You control how IOS opens stream connections to objects by specifying various open options in your IOS_$CREATE or IOS_$OPEN system call. For example, you can open an object permitting write access to the stream by specifying IOS_$WRITE_OPT. Most of these options determine how your program can access an object, and how programs from other processes can access an object. Section 4.3.7 describes these options in detail.

Table 4-5 lists the IOS_$OPEN_OPTIONS_T option set that control how IOS_$CREATE or IOS_$OPEN opens streams to objects.

Table 4-5. Options That Control how to Open Streams

Specifying this open option:	Causes the open call to:
IOS_$NO_OPEN_DELAY_OPT	Return immediately, instead of waiting for the call to complete.
IOS_$WRITE_OPT	Permit writing data to a new object. If a program tries to write on a stream for which you have not specified this option, it returns an error status. Note that when creating an object, this value is automatically set because the IOS manager assumes that when you create an object, you will want to write to it. Therefore, you do not need to specify this option on an IOS_$CREATE call.
IOS_$UNREGULATED_OPT	Permit unregulated (shared) concurrency mode. See Section 4.3.7 for details.
IOS_$POSITION_TO_EOF_OPT	Position the stream marker at the end of the object (EOF). Use this to append data at the end of an object.

Table 4-5. Options That Control how to Open Streams

Specifying this open option:	Causes the open call to:
IOS_$INQUIRE_ONLY_OPT	Open the object for attribute inquiries only; do not permit reading or writing of data.
IOS_$READ_INTEND_WRITE_OPT	Open the object for read access, with the intent that it can later be changed to write access. This allows other processes to read the object; but they cannot have write or read-intend-write access. See section 4.3.7 for details.

4.3.7. Controlling a Stream's Access and Concurrency

When you open a stream to an object, you determine how *your* program can use that object by specifying the stream's **access type**. At the same time, you determine how *other processes* can use the object by specifying the stream's **concurrency mode**. (You control a stream's access type and concurrency mode by specifying the appropriate open options in IOS_$OPEN_OPTIONS_T format.)

A stream's access type can be either read, write, or read-intend-write (RIW). Read and write access mean, respectively, that you allow your program to read from the object and write to the object. RIW access means that you currently allow your program to read from the object stream, and that you intend to change your program's access to write access in the future.

A stream's concurrency mode can be either regulated (protected) or unregulated (shared). Regulated concurrency mode means that you do *not* allow other programs read or write access to the object at the same time. Unregulated concurrency mode means that other programs can access the object at any time.

Together, the access type and concurrency mode allow you to determine how the object can be used. For example, if you open a stream to an object with *write access* and *regulated concurrency mode* (by specifying the IOS_$WRITE_OPT open option) only your program can access the object. Other processes that try to open a stream to the object will get the error, "Requested object is in use." However, if you open a stream to an object with *write access*, and *unregulated concurrency mode*, another process will be able to open a stream to the object, and can have any kind of access.

By specifying different combinations of access types and concurrency modes, you have a variety of ways to control how an object is used. Some DOMAIN managers refer to the combination of access type and concurrency mode as a **lock**. Also, some managers refer to the concurrency mode as being either **protected** or **shared**. That is, the object is either protected from other processes, or it is shared by other processes. The terms are analogous to the IOS manager's *regulated* and *unregulated* concurrency mode.

How you specify the type of access and concurrency mode when opening an object depends on how you expect to use the object. The following are some guidelines for determining access type and concurrency mode. Table 4-6 tells you which open options you can specify to get these combinations. Use **read** access and **regulated** concurrency mode when you expect several programs to read the object, but no program will write to the object.

Use **read-intend-write** (RIW) access and **regulated** concurrency mode when you want to read an object, and expect that you will write to it later. By doing this, you do not block other processes from reading the object, but they cannot write to the object. You can change the access to write when no other programs are reading it.

The Display Manager uses regulated RIW when it allows a user to edit an object. It opens the object for RIW, which allows the user to make edits to the object. At this time, other programs can read the object in its pre-modified form. When the user types CTRL/Y to close the object, the Display Manager changes the stream to write access and writes the changes to the disk.

Use **read** access and **unregulated** concurrency mode when you want to read an object, but also allow other programs on your node to write to the object (by getting shared write locks). You must synchronize the programs to handle reading and writing to the same object.

Use **write** access and **regulated** concurrency mode when you want to write to an object, and you want to deny any programs access to the object while you are writing.

Use **write** access and **unregulated** concurrency mode when you want to allow many programs to read from and write to an object. Note that you must synchronize the programs to handle concurrent reading and writing to the same object. (For details on synchronization techniques, see the *Programming with System Calls for Interprocess Communication* manual.)

Only programs on the same node can have unregulated write access to the same object, because they share the same physical memory for the object. When programs on different nodes share the same object, each node stores the object in its own memory. For this reason, programs on different nodes can have only unregulated read access, not unregulated write access.

Table 4-6 shows the predefined values that you can specify to get the type of control you want. These values are in IOS_$OPEN_OPTIONS_T format. The first column lists the combination (or lock) that you want. The second column lists the option (or options) you would specify on the open call to get the corresponding access type and concurrency mode. Note that the IOS manager assumes that most programs open objects using read access and protected concurrency mode. So, you don't need to specify these values in the open call.

Table 4-6. IOS Options for Specifying Access Types and Concurrency Modes

Combination	IOS Options to Specify
Regulated Read (Protected Read)	The empty set, []
Regulated RIW (Protected RIW)	[IOS_$READ_INTEND_WRITE_OPT]
Regulated Write (Exclusive Write)	[IOS_$WRITE_OPT]
Unregulated Read (Shared Read)	[IOS_$UNREGULATED_OPT]
Unregulated Write (Shared Write)	[IOS_$WRITE_OPT, IOS_$UNREGULATED_OPT]

Just as you set the concurrency mode to control how other processes can access the object you open, other processes will try to control how your program accesses the objects that it opens. If another process has already opened a stream to an object, and you try to open the same object with an incompatible access type and concurrency mode, then your open call will fail with the error, IOS_$CONCURRENCY_VIOLATION.

Refer to the following rules to determine whether the object you plan to open has compatible access types and concurrency modes with an existing open stream to the object.

If another process has opened the object for:

- Read access, regardless of the concurrency mode, you can open another stream for read or read-intend-write (IOS_$READ_INTEND_WRITE_OPT) access.

- Write access (IOS_$WRITE_OPT), and regulated (protected) concurrency, you cannot open another stream to the object.

- Write access (IOS_$WRITE_OPT), and unregulated (IOS_$UNREGULATED) concurrency, you can open another stream to the object for unregulated concurrency, regardless of the access type.

- Unregulated (IOS_$UNREGULATED) concurrency, regardless of the access, you can open another stream for unregulated concurrency -- as long as you open the object on the *same* node.

Table 4-7 summarizes the various access type and concurrency mode combinations that you can have.

Table 4-7. Access/Concurrency Combinations for Shared Streams

If another process opened a stream with:	You can open a stream to that same object with:	
Combination	**Access Type**	**Concurrency Mode**
Regulated Read	Read or RIW	Either mode
Regulated RIW	Read	Either mode
Unregulated Read	Read or RIW *or* Write	Either mode Shared *only*
Unregulated RIW	Read *or* RIW or Write	Either mode Shared *only*
Regulated Write	Cannot open another stream.	
Unregulated Write	Read, RIW, or Write	Shared *only*

4.3.8. Example of Controlling an Object's Access and Concurrency

Example 4-3 is a sample Pascal program that shows how to make sure that an object has compatible access and concurrency modes. Since the above rules state that only one object can be open with write access, the program must anticipate that its open call can fail if another process has an open stream to the object. Therefore, it tests for this error.

```
.
.
.
{ Open the object with write access. }
    done := FALSE;
    WHILE (done = FALSE)  DO
    BEGIN

         stream_id :=  ios_$open (pathname,
                                  namelength,
                                  [ios_$write,
                                  ios_$position_to_eof_opt],    { Append data }
                                  status);

         IF status.all = status_$ok THEN
              done := TRUE
         ELSE IF (status.all = ios_$concurrency_violation) THEN
         BEGIN
              writeln;
              writeln ( ' Can''t get object for write access.' );
              writeln ( ' Type YES if you want to try again. ');
              writeln ( ' Type NO to terminate program. ');
              readln (ans);
              IF (ans = 'NO') OR (ans = 'no') THEN
              BEGIN
                   done := TRUE;
                   writeln;
                   writeln (' Terminating program. ');
                   pgm_$exit;
              END;
         END
         ELSE IF (status.all <> status_$ok) THEN
         BEGIN
              error_$print( status );
              pgm_$exit;
         END;
    END; { while not done }
.
.
.
```

Example 4-3. Checking for Compatible Access Type and Concurrency Modes

4.4. Reading and Changing Object Attributes

When you create or open an object, the object has an associated set of attributes. These attributes fall into three categories: object, connection, and manager.

Object attributes describe an object's characteristics. For example, an object can contain ASCII data, or use FORTRAN carriage control characters. Table 4-8 lists the attributes associated with an object. Table 4-9 listes the FORTRAN carriage control characters.

Table 4-8. Object Attributes

Attribute	The object:
IOS_$OF_DELETE_ON_CLOSE	Will be deleted when all its associated streams close.
IOS_$OF_SPARSE_OK	Can be written as a sparse object.
IOS_$OF_ASCII	Contains ASCII data.
IOS_$OF_FTNCC	Uses FORTRAN carriage control characters.*
IOS_$OF_COND	Has get or put calls performed conditionally, as if the IOS_$COND_OPT was specified on a get or put call.

* In the FORTRAN carriage control format, the first character of each record is a carriage control character. The characters listed in Table 4-9 are recognized as FORTRAN carriage control characters; all others are ignored. Each line must end with a NEWLINE character.

Table 4-9. FORTRAN Carriage Control Characters

Character	Effect
space	Go to beginning of next line.
0	Skip one line.
1	Skip to beginning of next page.
+	Overprint: go to beginning of current line.

Connection attributes describe the characteristics of a specific stream connection. For example, a stream can behave like a Display Manager pad, or it can be written. Stream connection attributes affect the behavior of a *single* stream only, so two streams open to the same object can have different connection attributes. Table 4-10 lists the attributes associated with a stream connection.

Table 4-10. Stream Connection Attributes

Attribute	**The connection:**
IOS_$CF_TTY	Behaves like a terminal.
IOS_$CF_IPC	Behaves like an interprocess communication (IPC) channel.
IOS_$CF_VT	Behaves like a DOMAIN Display Manager pad.
IOS_$CF_WRITE	Can be written.
IOS_$CF_APPEND	Positions its stream marker to the end of the object (EOF) before each put call.
IOS_$CF_UNREGULATED	Is open for unregulated (shared) concurrency mode.
IOS_$CF_READ_INTEND_WRITE	Is open for read access, and can later change to write access.

Manager attributes describe the operations that a type manager will allow to be performed on that type of object. For example a type manager might allow programs to create objects of this type or use different record formats. Table 4-11 lists the attributes associated with a type manager.

Even if the type manager permits an operation, a specific object of that type might not be able to perform the operation. Consider, for example, the write operation that allows writing to sparse objects. (A **sparse** object is an object that can contains gaps created when a program seeks past EOF and then writes to the object.) Both the type manager's and the object's attribute set must contain the appropriate attribute to permit writing to sparse objects before the operation can actually be allowed.

You set some of the object attributes when you create an object. You set connection attributes by specifying certain open options in the create or open call. For example, if you open an object specifying the IOS_$WRITE_OPT, the object's stream connection set will contain the IOS_$CF_WRITE attribute.

You can add attributes to either the object or stream connection set after opening the object with the IOS_$SET_CONN_FLAG or IOS_$SET_OBJ_FLAG calls. Section 4.4.1 describes how to use these calls. Section 4.4.2 is a program segment using the IOS_$INQ... and IOS_$SET... calls.

Table 4-11. Type Manager Attributes

Attribute	The type manager can:
IOS_$MF_CREATE	Create other objects.
IOS_$MF_CREATE_BAK	Create backup (.bak) objects.
IOS_$MF_IMEX	Export streams to new processes.
IOS_$MF_FORK	Pass streams to forked processes.
IOS_$MF_FORCE_WRITE	Force-write object contents to disk.
IOS_$MF_WRITE	Perform write operations.
IOS_$MF_SEEK_ABS	Perform absolute seeks.
IOS_$MF_SEEK_SHORT	Perform seeks using short (4-byte) seek keys.
IOS_$MF_SEEK_FULL	Perform seeks using full (8-byte) seek keys.
IOS_$MF_SEEK_BYTE	Perform seeks to byte positions.
IOS_$MF_SEEK_REC	Perform seeks to record positions.
IOS_$MF_SEEK_BOF	Perform seeks to the beginning of the object.
IOS_$MF_REC_TYPE	Support various record type formats.
IOS_$MF_TRUNCATE	Truncate object.
IOS_$MF_UNREGULATED	Have unregulated (shared) concurrency mode.
IOS_$MF_SPARSE	Support sparse objects.
IOS_$MF_READ_INTEND_WRITE	Have RIW access.

4.4.1. Inquiring about and Changing Object Attributes

You can use the following IOS calls to determine an object's current object, connection and manager attribute sets: IOS_$INQ_OBJ_FLAGS, IOS_$INQ_CONN_FLAGS, and IOS_$INQ_MGR_FLAGS.

Typically, you would use these calls directly after opening an object to determine what types of operations can be performed on that object. If the object, connection, or manager set has the attribute, the set contains the value.

You initially set object or connection attributes when you create or open an object. A type manager sets the attributes for the manager set when it implements the type operations. You can change the initial object or connection attribute set by using the IOS_$SET_OBJ_FLAG or IOS_$SET_CONN_FLAG, respectively.

Note that the attribute set does not list the read access or regulated concurrency as values in the set. Rather, *all* stream connections have these two qualities, so the IOS manager does not consider them as attributes that you can add or subtract from a set.

Add attributes to the object or connection attribute set with the IOS_$SET_OBJ_FLAG or IOS_$SET_CONN_FLAG, respectively. Specify the desired attribute in the second parameter of either call, and a value of TRUE in third parameter. To remove attributes from either set, specify the attribute and a value of FALSE. Note that you must make a separate call to add or remove each attribute from its respective set.

After changing the attribute set, you can perform another IOS_$INQ to see the full attribute set. Note that you might have what appears to be conflicting values in the set. For example, if you open the object with RIW access, and then change the access to write, the attribute set will contain both RIW and write attributes (unless you explicitly removed RIW from the set).

If the object connection set contains both the RIW and write access attributes, the stream connection has write access.

This is useful when you want the object to be available for read access most of the time, and you plan to write to the object for only short intervals. You can open the object for RIW access, and then change it to write access by setting IOS_$CF_WRITE to TRUE when writing to the object. You can change the access back to RIW by simply setting IOS_$CF_WRITE to FALSE. Since the RIW attribute is still in the set, the object has RIW access.

4.4.2. Example of Inquiring about and Changing Attributes

The program in Example 4-4 uses the IOS_$INQ calls to get the object and manager set of attributes for an object. This program uses the DOMAIN Pascal functions FIRSTOF and LASTOF (which are extensions to ISO/ANSI Standard Pascal) to get the first and last possible value in each set of object attributes.

```
PROGRAM ios_inq_attributes;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/type_uids.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR

  status : status_$t;
  count  : integer;
  ans    : string;
```

Example 4-4. Inquiring About an Object

```
  {$CREATE variables}
  pathname     : name_$pname_t;
  namelength   : integer;
  type_uid     : uid_$t;
  create_mode  : ios_$create_mode_t;
  open_opt     : ios_$open_options_t;
  stream_id    : ios_$id_t;

  {INQ_FLAGS variables}
  conn_flags : ios_$conn_flag_set;
  obj_flags  : ios_$obj_flag_set;
  mgr_flags  : ios_$mgr_flag_set;
  c_flg      : ios_$conn_flag_t;
  o_flg      : ios_$obj_flag_t;
  m_flg      : ios_$mgr_flag_t;

PROCEDURE check_status;  { for error handling }
BEGIN    {main}

    { Ask user for pathname and convert it to internal format using VFMT. }
    .
    .
    .
    { Create the object. }
    ios_$create (pathname,
                 namelength,
                 uasc_$uid,               { Unstructured ASCII Type UID }
                 ios_$no_pre_exist_mode,  { Return error if exists }
                 [ios_$write_opt,         { Permit write access }
                  ios_$unregulated_opt],  { Permit concurrent users }
                 stream_id,
                 status);
    check_status;

    { Get object attributes with IOS_$INQ_OBJ_FLAG. }

    obj_flags := ios_$inq_obj_flags (stream_id,
                                     status);
    check_status;

    writeln;
    writeln ('Object Attributes of Created Object:');
    writeln;

    { Write each attribute in the set. }
    FOR o_flg := firstof( ios_$obj_flag_t ) TO
        LASTOF( ios_$obj_flag_t ) DO
            IF o_flg IN obj_flags THEN
                writeln( '    ', o_flg );
```

Example 4-4. Inquiring About an Object (Cont.)

```
    { Get manager attributes with IOS_$INQ_MGR_FLAG. }
    mgr_flags := ios_$inq_mgr_flags (stream_id,
                                     status);
    check_status;

    writeln;
    writeln ('Manager Attributes of Created Object:');

    { Write each attribute in the set. }
    FOR m_flg := FIRSTOF( ios_$mgr_flag_t ) TO
        LASTOF( ios_$mgr_flag_t ) DO
            IF m_flg IN mgr_flags THEN
                writeln( '    ', m_flg );

    { Get connection attributes with IOS_$INQ_CONN_FLAG. }
    conn_flags := ios_$inq_conn_flags (stream_id,
                                       status);
    check_status;
    .
    .
    .
END. {ios_inq_set_attributes }
```

Example 4-4. Inquiring About an Object (Concluded)

4.4.3. Example of Changing Attributes

Example 4-5 is a sample Pascal program that changes an object attribute set from RIW to write access. The program opens an object with RIW access so that other programs can read the object until it needs to write to the object.

Since the program cannot change the access to write until no other processes have the object open, the program keeps trying until it can. The program uses IOS_$SET_CONN_FLAG to add write access to the object's attribute set. Note that the set still contains RIW access, because the program did not explicitly remove this attribute. This way, the program allows other processes to read the object by simply removing the write access attribute from the set as soon as it finishes writing to the object.

```
PROGRAM ios_riw_to_write;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/cal.ins.pas';
```

Example 4-5. Changing an Object from RIW to Write Access

```
VAR
  status    : status_$t;
  count     : integer;
  ans       : string;
  rel_time  : time_$clock_t;
  done      : boolean;

  {$OPEN variables}
  pathname     : name_$pname_t;
  namelength   : integer;
  open_opt     : ios_$open_options_t;
  stream_id    : ios_$id_t;

  {INQ_FLAGS variables}
  conn_flags : ios_$conn_flag_set;
  c_flg      : ios_$conn_flag_t;

  { OPEN variables }
  msg        : string := 'Writing to the object. ';

BEGIN    {main}

{ Ask user for filename convert it to internal format using VFMT. }

{ Open the object with RIW access. }

    stream_id :=  ios_$open (pathname,
                             namelength,
                             [ios_$read_intend_write_opt,
                              ios_$position_to_eof_opt],   { Append data }
                             status);
    check_status;

{ Add write access to the object's connection attribute set so it
  can write to the object. If it cannot change the object's access,
  it keeps trying until it does, or until user types NO.
  Try locking object, if it can't, send messsage to user. }
```

Example 4-5. Changing an Object from RIW to Write Access (Cont.)

```
    done := FALSE;
    WHILE (done = FALSE)  DO
    BEGIN
        ios_$set_conn_flag (stream_id,
                            ios_$cf_write,
                            TRUE,            { Add write access to set }
                            status);

         IF status.all = status_$ok THEN
              done := TRUE
         ELSE BEGIN
              writeln;
              writeln ( ' Cant lock object for writing.' );
              writeln ( ' Type YES if you want to try again. ');
              writeln ( ' Type NO to terminate program. ');
              readln (ans);
              IF (ans = 'NO') OR (ans = 'no') THEN
              BEGIN
                   done := TRUE;
                   writeln;
                   writeln (' Terminating program. ');
                   pgm_$exit;
              END;
         END;

    END; { while not done }

{ Write message to the object. }

    ios_$put ( stream_id,          { Stream ID }
               [ios_$cond_opt],    { Default put options }
               msg,                { Buffer to hold message }
               SIZEOF(msg),        { Length of message  }
               status);
    check_status;

{ Write message to user. }

    IF status.all = status_$ok THEN
         writeln ('Wrote message to object.');

{ Remove write access from set, so other processes can open the
  object for read access again. }

    ios_$set_conn_flag (stream_id,
                        ios_$cf_write,
                        FALSE,           { Remove write access }
                        status);

    check_status;
END. { ios_riw_to_write }
```

Example 4-5. Changing an Object from RIW to Write Access (Concluded)

4.4.4. Getting Additional Information about Objects and Directories

The IOS manager provides a few calls to get additional information about an object. Table 4-12 lists these calls.

Table 4-12. Getting Additional Information about an Object

IOS Call	Description
IOS_$INQ_FILE_ATTR	Returns an object's usage attributes: date and time created, date and time last used, date and time last modified, and number of blocks in the object.
IOS_$INQ_PATH_NAME	Returns the pathname of an object open on a specified stream. The pathname can be in any one of the following formats: absolute pathname from the root (//) directory; name relative to the root, working, naming or "node_data" directory; or the or residual name if stream was opened using extended naming.
IOS_$INQ_TYPE_UID	Returns the type UID of an object.

The IOS manager also provides a call to determine or set your current working or naming directory. IOS_$GET_DIR returns the current working or naming directory. IOS_$SET_DIR changes the current working or naming directory to the pathname you specify in the first parameter of the call.

4.5. Closing and Deleting Objects

Although the system automatically closes the streams your program opens when the program terminates, it is good practice to close the streams explicitly with IOS_$CLOSE. This way you can also report any errors that occur during the close operation.

To close a stream to an object, call IOS_$CLOSE and specify the stream ID of the open stream. Your program can close only those streams that it has opened at the current or lower program levels (that is, streams opened by programs that the calling program invoked). IOS_$CLOSE returns an error if you try to close a stream in the current program that was opened by its invoker.

You can make a permanent copy of the object without closing the stream by calling IOS_$FORCE_WRITE_FILE. Use this call to ensure that the object is stored safely in the event of a system crash. Safe storage depends on the object type. For most object types, safe storage is the disk. Safe storage for a magnetic tape descriptor object is the tape.

If you have completed processing an object and have no further need for it, you should delete it. To delete an object, call IOS_$DELETE, specifying the stream ID of the open object. If more than one stream is open to the object, IOS_$DELETE *marks* the object for deletion, but the object still exists until *all* streams to the object are closed.

The IOS_$DELETE call actually sets the delete-on-close object attribute (IOS_$OF_DELETE_ON_CLOSE) to TRUE, then closes the stream. So, if the type manager does not allow the object to have the delete-on-close attribute, the delete call fails. In this case, the call closes the stream but does not delete the object.

You can also use IOS_$TRUNCATE to delete the contents of an object following the current stream marker.

4.6. Writing to Objects

Use the IOS_$PUT call to write data to any kind of object. Specify the stream ID of the open stream you want to write the data to, a buffer containing the data, and the size of the buffer. You can also specify various put options, in IOS_$PUT_GET_OPTS_T format, depending on the type of object you are writing to.

Table 4-13 lists the put options in IOS_$PUT_GET_OPTS_T format that you can specify in an IOS_$PUT call.

Table 4-13. Options to Control an IOS_$PUT call

Put Option	Description
IOS_$COND_OPT	Writes data only if it can be done without waiting. If the put call must wait, it returns the IOS_$PUT_CONDITIONAL_FAILED error status. A call would have to wait if the receiver was *full*, for example, a mailbox couldn't hold any more messages.
IOS_$PREVIEW_OPT	Writes data but does not update the stream marker.
IOS_$PARTIAL_RECORD_OPT	Writes a portion of a record but does not terminate it. IOS_$PUT terminates the record when you call IOS_$PUT without specifying this option. If you do not specify this option, IOS_$PUT writes a full record. You can use this option with record-oriented objects only. Type managers that do *not* support records ignore this option. For information on record-oriented objects, see Section 4.10.

4.6.1. Example of Writing to Objects

The program in Example 4-6 shows how to write data using IOS_$PUT. The program writes to a UASC object type, line by line. To store and retrive data by lines, the program explicitly embeds NEWLINE characters at the end of each line of input. To embed a NEWLINE character in a UASC object, use the CHR Pascal function to assign the ASCII NEWLINE character value (which is 10) to a byte at the end of the line buffer array. (You can also use the PAD_$NEWLINE constant instead of CHR.)

This program asks the user to type data into a UASC object line by line. It then performs the following:

- Defines an input buffer, "line," as a character array. This buffer holds the data that you want to write.
- Calls IOS_$CREATE to create a new, or open an existing UASC object.
- Loads the buffer, using input from the user.
- Calculates the length of the line.
- Terminates the line with a NEWLINE character.
- Writes the line, using IOS_$PUT.

```
PROGRAM ios_put_uasc_newline;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  status : status_$t;
  count  : integer;

  { $CREATE variables }
  pathname     : name_$pname_t;
  namelength   : integer;
  stream_id    : ios_$id_t;

  { $PUT variables }
  line     : string;
  linelen  : integer;

BEGIN { main }

    { Get the pathname and convert it to internal format using VFMT.  }
    .
    .
    .
    { Create the object, or open an existing object for appending input. }
    ios_$create (pathname,
                 namelength,
                 uid_$nil,                   { UASC type UID }
                 ios_$preserve_mode,         { Open object if exists }
                 [ios_$position_to_eof_opt], { Append data }
                 stream_id,
                 status);
    check_status;
```

Example 4-6. Writing to a UASC Object Line by Line

```
    { Get a line of input from keyboard. }
    writeln ('Type in a line or CTRL/Z to stop:');
    WHILE NOT eof DO
    BEGIN
        { Load keyboard input into buffer. }
        readln(line);
        linelen := SIZEOF(line);

        WHILE (line[linelen] = ' ') AND (linelen > 0) DO
            linelen := linelen - 1;

        { Terminate line with NEWLINE character. }
        linelen := linelen + 1;
        line[linelen] := CHR(10);

        { Write the line to a object. }
        ios_$put ( stream_id, { Stream ID }
                   [],        { Default put options }
                   line,      { Buffer to hold input line }
                   linelen,   { Length of line  }
                   status);
        check_status;

        writeln ('Type in another line or CTRL/Z to stop:');
    END;{ while not EOF }
END. { ios_put_uasc_newline }
```

Example 4-6. Writing to a UASC Object Line by Line (Concluded)

4.7. Reading Objects

The IOS manager supplies the following two calls for reading data from objects:

IOS_$LOCATE Reads data from a stream and returns a pointer to the data.

IOS_$GET Reads data from a stream and copies the data into a buffer.

Regardless of whether you use IOS_$LOCATE or IOS_$GET, we refer to this as the **get call**.

In most cases, use IOS_$LOCATE to read data because it is faster, since it does not perform a copy operation while reading. One drawback to using IOS_$LOCATE is that the pointer that IOS_$LOCATE returns is valid only until the next IOS call. If you cannot tolerate this drawback, use IOS_$GET. For example, you would use IOS_$GET when you need to read more data than can be obtained in one call -- like when you need to read and rearrange a number of lines from an object.

Normally, IOS_$LOCATE locates data and returns a pointer to the data. However, not all managers support the internal buffering necessary for IOS_$LOCATE to work this way. In these cases, IOS_$LOCATE will not be able to return a pointer to the data. Instead, IOS_$LOCATE actually creates a buffer and then calls IOS_$GET to perform the get call. If this occurs, IOS_$LOCATE is no more efficient than IOS_$GET. The size of the buffer that IOS_$LOCATE creates is either the length you specify in "data-size," or 1024 bytes, whichever is the smaller. You can use the IOS_$SET_LOCATE_BUFFER_SIZE call to specify a buffer larger than 1024 bytes, if necessary.

You can control how the IOS get call reads data by specifying any of the get options listed in Table 4-14.

Table 4-14. Options to Control an IOS Get Call

Get Option	Description
IOS_$COND_OPT	Reads data, if available. Use this option to read data from places where it might not be available immediately, for example, SIO lines, mailboxes, and input pads. IOS_$GET returns the IOS_$GET_CONDITIONAL_FAILED status code if data is not available, and sets the return value of "ret-length" to 0.
IOS_$PREVIEW_OPT	Reads data but does not update the stream marker.
IOS_$NO_REC_BNDRY_OPT	Ignores record boundaries while reading data. For example, it ignores NEWLINE characters in a UASC object, which guarantees that the call fills the specified buffer.

When an IOS get call returns either a pointer to the data (IOS_$LOCATE) or a buffer containing the data (IOS_$GET), it also returns the amount of data read, in the return value, "return-length." You can specify how much data to read with the input parameter, "buffer-size." If the get call reads the data successfully, the "return-length" equals the amount of data read. If the get call does not return any data, "return-length" equals the value, 0.

If you did not specify a large enough buffer for the returning data, the get call:

- Reads enough data to fill the requested size
- Sets "ret-length" equal to "buffer-size"
- Positions the stream marker to the first unread byte
- Returns the IOS_$BUFFER_SIZE_TOO_SMALL status code to indicate that this condition has occurred

You can inquire about how many bytes remain to be read in the current record by calling IOS_$INQ_REC_REMAINDER.

There are two methods for accessing data from objects: sequential access and random access. In **sequential access**, multiple get calls read an object from beginning to end of the object. That is, a program using sequential access reads the first line, then the second, and so on.

In **random access**, the get call reads objects from a object in random fashion. For example, a program using random access might read byte position 12, then byte position 7, and so on.

The following sections describe how to get data from an object using both methods.

4.8. Reading Objects Sequentially

Sequential access occurs when the get call reads an object from the beginning to the end. Each get call reads a specified amount of data at a time, for example, one line, or one record or 4 bytes. You specify the amount of data you want to read in the fourth parameter of the get call. Since the get call returns a fixed amount of data per call, you can simply use it within a loop to read more data. In most cases, the loop reads data until it reads the end of object (EOF) marker.

The program in Example 4-7 asks the user to specify an existing UASC object, and then reads the object sequentially. The program does the following:

- Declares a constant to indicate how much data you want to read. If this is smaller than the amount of data to read, the get call returns the IOS_$BUFFER_TOO_SMALL error.
- Declares a pointer to the string that contains the data to be read.
- Opens the existing object that the user specified with IOS_$OPEN.
- Enters a loop that:
    1. Reads a line from the object using IOS_$LOCATE.
    2. Tests for the IOS_$END_OF_FILE, and other get call errors.
    3. Writes the line to standard output by specifying values returned by IOS_$LOCATE: the amount of data read, and the pointer that points to the data. Note that it must dereference the pointer variable.
- Exits the loop when the get call reads an EOF.

```
PROGRAM ios_locate;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/type_uids.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

CONST
  data_size = 1024;             { Amount of data to read }

VAR
  status : status_$t;
  count  : integer;
  pathname    : name_$pname_t;
  namelength  : integer;
  open_opt    : ios_$open_options_t;
  stream_id   : ios_$id_t;
```

Example 4-7. Reading Sequentially From an Object

```
  { $GET variables }
  ret_length    : integer32;      { Amount of data read }
  line          : string;         { String containing line read }
  data_ptr      : ^string;        { Pointer to returned data }

BEGIN    { main }

    { Get the pathname and convert it to internal format using VFMT. }
    .
    .
    .
    { Open the object. }
    stream_id :=  ios_$open (pathname,
                             namelength,
                             [ios_$read_intend_write_opt], { RIW access }
                             status);
    check_status;

    WHILE (status.all = status_$ok)  DO
    BEGIN

        { Read data until an EOF is encountered. Set the IOS_$COND_OPT
          option, in case data is not available immediately. }
        ret_length := ios_$locate (stream_id,
                                   [ios_$cond_opt],
                                   data_ptr,
                                   data_size,
                                   status);

        { Test for read errors. }
        IF status.all = ios_$end_of_file THEN
            writeln (' End of file reached. ');
        IF status.all = ios_$buffer_too_small THEN
           vfmt_$write2 ( '%d byte buffer too small on stream %wd%.',
                          data_size, stream_id)
        ELSE IF (status.all = ios_$get_conditional_failed) THEN
           writeln (' No data available.')
        ELSE IF (status.all <> status_$ok) THEN
            check_status;

        { Write data to standard output by dereferencing
          the pointer that points to the line read. }

        ios_$put ( ios_$stdout,
                   [],
                   data_ptr^,  { Dereference pointer }
                   ret_length, { Amount returned by IOS_$LOCATE }
                   status);
        check_status;
    END; { While not EOF }

    { Close the stream of the open object before terminating. }
    ios_$close (stream_id,
                status );
END. { ios_locate }
```

Example 4-7. Reading Sequentially From an Object (Concluded)

4.9. Performing Random Access

Random access is the method by which an object is read (and processed) nonsequentially. For example, a get call can read starting at byte position 12, then byte postion 7, then byte position 41.

To access an object randomly, you perform one of the IOS_$SEEK calls to reposition the stream marker to a specified location. Then, you perform a get call.

The IOS manager provides two kinds of seek operations: nonkeyed and keyed. In a **nonkeyed** seek a program moves the stream marker to:

- The beginning or end of the object
- A specified byte position
- A specified record position

In a **keyed** seek, a program stores and retrieves information by identifying positions on a **seek key**.

Whether you perform a nonkeyed or keyed seek depends on how the object's data is represented. For example, programs that need perform "arithmetic" on the data (such as comparing two positions) will use nonkeyed seek operations. Programs that require only the ability to move from one position to another in an object will use keyed seek operations.

The following sections describe the two types of seeks.

4.9.1. Nonkeyed Seeking

You can perform a nonkeyed seek on an object by specifying the beginning or end of the object, or any offset from the beginning of the object.

To move the stream marker to the beginning of the object, call IOS_$SEEK_TO_BOF. To move the stream marker to the end of the object, call IOS_$SEEK_TO_EOF.

To obtain the offset of the stream marker, use IOS_$INQ_BYTE_POS or IOS_$INQ_REC_POS. (Use the latter if your object is record-oriented.) These calls return the current position of the stream marker from the beginning of the object. The calls can also return the position of the stream marker at the beginning of the object (which is always 0), or the end of the object (which indicates the length of the object in bytes or records).

Once you have the returned offset, you can move the stream marker to the desired location by calling IOS_$SEEK. You can continue to move the stream marker to offsets from the beginning, or end of the object -- this is called **absolute** seeking. Or you can move the stream marker to offsets from the current position -- this is called **relative** seeking.

4.9.2. Keyed Seeking

Keyed seeking is based on positioning information that the IOS manager provides with a seek key. You get a seek key by using either IOS_$INQ_FULL_KEY or IOS_$INQ_SHORT_KEY. These calls return a value that represents the position of the stream marker at the time of a call. By storing this returned seek key, you can return to the position at a later time.

Whether you get a **full** seek key or **short** seek key depends on your application program. A full seek key is 8-bytes long and represents an exact stream position. A short seek key is 4 bytes long and represents a stream position up to a record boundary. Since short seek keys require half the storage space as full seek keys, you might want to use short seek keys if your application program stores a large number of seek keys. However, short seek keys are limiting in that you can only indicate *record* boundary positions, while full seek keys allow you to indicate *any* position.

Use seek keys merely as an index -- do not depend on their *contents*. The contents of a seek key remains private to the IOS manager, which guarantees only that the seek key returns to the position it describes.

4.9.3. Example of Using Seek Keys

The program in Example 4-8 uses seek keys to access lines (by line number) randomly in a UASC object. Note that a line number is not the same thing as a record number.

The program does the following:

- Declares a seek-key vector to store seek key values. Since it is using short seek keys, this is an array of 4-byte integers.
- Opens a UASC object.
- Enters a loop to read the object sequentially. The program:
    1. Gets a seek key by calling IOS_$INQ_SHORT_KEY.
    2. Reads a line.
    3. Stores the returned seek key in the array of seek keys. Note that by doing this, the vector is indexed by line number.
- Prompts the user for a line number.
- Moves the stream marker to the desired line by calling IOS_$SEEK_SHORT_KEY. This call associates the seek key with the line number that the user specified.
- Reads the line by calling IOS_$LOCATE.
- Writes the line to output and continues to prompt the user until the user types a CTRL/Z to stop.

```
PROGRAM ios_seek_uasc;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

CONST
  max_lines  = 1024; { Maximum number of lines in object }

VAR
  status : status_$t;
  count  : integer;

  {$OPEN variables}
  pathname    : name_$pname_t;
  namelength : integer;
  stream_id     : ios_$id_t;

  { $GET variables }
  line         : string;
  ret_len      : integer32;
  choice_line : integer;
  no_of_lines : integer;

  { $SEEK variables }
  short_key   : integer32;

  { Declare vector to hold seek keys }
  seek_vector : ARRAY[1..max_lines] OF integer32;

BEGIN { main }

    { Get the pathname and convert it to internal format using VFMT. }
    { Open the object for reading. }
    { Read the object and fill the seek_vector with seek keys. }
    .
    .
    .
    no_of_lines := 0;
    WHILE status.all = status_$ok DO
    BEGIN { while there is data in object }

        { Get a short seek key. }
        short_key := ios_$inq_short_key (stream_id,
                                         ios_$current,  { position }
                                         status);
        check_status;
```

Example 4-8. Accessing a UASC Object Randomly Using Seek Keys

```
    { Read a line. }
    ret_len := ios_$get ( stream_id,
                          [],           { put-get options }
                          line,
                          SIZEOF(line),
                          status);
    { Test for EOF. }
    IF (status.all = ios_$end_of_file) THEN
        EXIT;
    check_status;

    { Increment the vector index. }
    no_of_lines := no_of_lines + 1;

    { Test for maximum number of lines. }
    IF no_of_lines <= max_lines THEN
        { Load vector with the returned seek key. }
        seek_vector[no_of_lines] := short_key
    ELSE
    BEGIN
        writeln('Maximum number of lines exceeded. ');
        EXIT;
    END; { IF no_of_lines <= max_lines }

END; { while there is data in object }

{ Prompt the user for a line number. }
write( 'Type the number line you want to see:');
writeln(' (1 - ', no_of_lines:1, ' or CTRL/Z to stop:');

WHILE NOT eof DO BEGIN { while user wants more }

    readln(choice_line);
    { Test to see if the chosen line is in range. }
    WHILE (choice_line <= 0) OR (choice_line > no_of_lines) DO
    BEGIN
        write ('Line number is out of range. Enter a number');
        writeln('  between 1 and ', no_of_lines:1, ':');
        readln(choice_line);
    END;

    { Load the seek key using the vector. }
    short_key := seek_vector[choice_line];
    ios_$seek_short_key ( stream_id,
                          short_key, {4-byte integer}
                          status);
    check_status;

    { Read the line. }
    ret_len := ios_$get ( stream_id,
                          [],           { put-get options }
                          line,
                          SIZEOF(line),
                          status);
    check_status;
```

Example 4-8. Accessing a UASC Object Randomly Using Seek Keys (Cont.)

```
        { Write the line to output. }
        writeln(line : ret_len);

        { Prompt for next line. }
        write( 'Type the next number line you want to see:')
        writeln(' (1 - ', no_of_lines:1, ' or CTRL/Z to stop:');

    END;{while}

END.  { ios_seek_uasc }
```

Example 4-8. Accessing a UASC Object Randomly Using Seek Keys (Concluded)

4.10. Handling Record-Oriented Object Types

The UASC object type *thinks* of data as flowing in a continuous stream. In contrast, the record-oriented object type *thinks* of data as being broken into discrete groups, or records. A record boundary marks the end of each record.

Get and put calls recognize these record boundaries. So, when using get and put calls on record-oriented objects, the calls return the data contained in a single record at a time, even if you request more data than is contained in the record.

For example, you have a record-oriented object whose first three records are 12-bytes, 16-bytes, and 32-bytes long. If you specify a buffer size of 16 bytes, three successive put calls would perform the following:

- The first put call returns the first record (12 bytes) because the record is smaller than the size of the buffer.

- The second put call returns the second record (16 bytes) because the record is equal to the buffer size.

- The third put call returns the error IOS_$BUFFER_SIZE_TOO_SMALL because the buffer is too small to hold the next record, which is 32-bytes long. (If this happens, you might use IOS_$INQ_REC_REMAINDER to determine the number of bytes in the record left to be read.)

You can use most IOS calls to operate on record-oriented objects. Some calls provide options particular to record-oriented objects. For example, the IOS_$PARTIAL_RECORD_OPT option on a put call allows you to write portions of a record without terminating it. Currently, DOMAIN supports the REC object type. Users can implement their own record-oriented object types by writing a type manager.

The following sections describe how to perform I/O using the two most common record formats, variable-length and fixed-length records. Section 4.10.1 describes how to write to fixed-length record objects. Section 4.10.2 describes how to write to variable-length record objects. Section 4.10.3 shows how to read data from fixed-length record objects using seek keys. Section 4.10.4 describes the possible record formats that a record-oriented type can have.

4.10.1. Writing Fixed-Length Records

To write to a object containing records you open an object specifying a type UID that handles records, such as DOMAIN's RECORDS_$UID.

The program in Example 4-9 asks the user to type data into a record-oriented object that contains employee records. It performs the following:

- Defines an employee information record "info_rec" containing fields for employee name, number, and address.
- Creates a record-oriented object, using IOS_$CREATE. (To handle fixed-length records, this program declares the record data type with fields of the same length.)
- Loads the record, using input from the user.
- Writes the record to the object, using IOS_$PUT.

```
PROGRAM ios_put_rec_fixed;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/type_uids.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

TYPE
  info_rec_t = RECORD    { Employee record }
      emp_id  : integer;
      address : string;
      name    : string;
  END;

VAR
  status : status_$t;
  count  : integer;

  { $CREATE variables }
  pathname     : name_$pname_t;
  namelength   : integer;
  stream_id    : ios_$id_t;

  { $PUT variables }
  line     : string;
  info_rec : info_rec_t;
```

Example 4-9. Writing Fixed-Length Records

```
BEGIN { main }

{ Get the pathname. }
    writeln ('Type the pathname of the object you want to create: ');
    namelength := SIZEOF(pathname);        { Max namelength }

{ Transfer the pathname into internal format using VFMT. }
    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);

{ Create the object, or open an existing object for appending input. }
    ios_$create (pathname,
                 namelength,
                 records_$uid,                 { Record Type UID  }
                 ios_$preserve_mode,           { Open object if exists }
                 [ios_$position_to_eof_opt],   { Append data }
                 stream_id,
                 status);
    check_status;

{ Get a line of input. }
    writeln ('Type employee name or CTRL/Z to stop:');
    WHILE NOT EOF DO
    BEGIN
        readln(info_rec.name);
        writeln('Type employee id #:');
        readln(info_rec.emp_id);
        writeln('Type address of employee on one line: ');
        readln(info_rec.address);

        { Write the record. }

        ios_$put ( stream_id,  { Stream-id of open object }
                   [],         { Put options }
                   info_rec,   { Data buffer }
                   SIZEOF(info_rec), { Length of data buffer }
                   status);    { Completion status }
        check_status;

        writeln;
        writeln (' Record written. ');
        writeln ('Type the next employee name or type CTRL/Z to stop:');

    END;{while}
END. { ios_put_rec_fixed }
```

Example 4-9. Writing Fixed-Length Records (Concluded)

4.10.2. Writing Variable-Length Records

You can write variable-length records to an object in the same way that you write fixed-length records to an object except, since the data buffer varies, you must calculate its length. A common way to do implement a variable-length buffer is to write to the variable-length field, calculate its length, then write the length in a field containing the length. To write to individual fields of a record, call IOS_$PUT with the IOS_$PARTIAL_RECORD_OPT put option. When you want to terminate the record, write the last portion of the record by using IOS_$PUT without specifying the IOS_$PARTIAL_RECORD_OPT option.

The program in Example 4-10 uses IOS_$PARTIAL_RECORD_OPT to write variable-length records. After the user types an employee name, the subroutine, PUT_NAME_LENGTH, calculates the length and puts that value in the record's "namelen" field.

Since the name field of this record varies in length, the records are of variable length. Note that, in Pascal, you must declare the variant portion of a record in the last field. Note also that you may not be able to handle variable-length records if your object type does not support them. See Section 4.10.4 for details.

This program performs the following:

- Defines an employee information record "info_rec" containing three fields: length of the employee name "namelen," the employee ID number "emp_id," and the employee name "name."

- Declares a procedure "put_name_length" to calculate the length of the input name, and writes the result to the output object separately, using the IOS_$PARTIAL_RECORD_OPT put option.

- Creates a record-oriented object by specifying the RECORDS_$UID type UID.

- Loads the record, using input from the user.

- Calls the "put_name_length" procedure to calculate the length of the employee's name and write the length into the first field of the record "namelen."

- Writes the second field "emp_id" of the record, using IOS_$PARTIAL_RECORD_OPT.

- Writes the last field of the record "name," using IOS_$PUT. This terminates the record because the program did *not* specify IOS_$PARTIAL_RECORD_OPT.

```
PROGRAM ios_partial_rec;

{ This program uses partial records. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/type_uids.ins.pas';

TYPE

  info_rec_t = RECORD          { Employee record }
      namelen  : integer;
      emp_id   : integer;
      name     : string;       { Variable-length field at end }
  END;

VAR
  status : status_$t;
  count  : integer;

  { $CREATE variables }
  pathname     : name_$pname_t;
  namelength   : integer;
  stream_id    : ios_$id_t;

  { $PUT variables }
  line      : string;
  info_rec  : info_rec_t;

PROCEDURE  put_name_length;

{ This procedure calculates the length of the employee name
  and puts the value into the namelen field. }

BEGIN

{ Calculate the length of info_rec.name. }
    info_rec.namelen := SIZEOF(info_rec.name);
    WHILE (info_rec.name[info_rec.namelen] = ' ') AND
          (info_rec.namelen > 0) DO
        info_rec.namelen := info_rec.namelen - 1;

{ Put the value of namelength into the record. }
    ios_$put ( stream_id,                   { Stream ID of open object }
               [ios_$partial_record_opt],  { Put options }
               info_rec.namelen,            { Data buffer }
               SIZEOF(info_rec.namelen),    { Length of data buffer }
               status);
    check_status;

END;{put_name_length}
```

Example 4-10. Writing Variable-Length Records

```
BEGIN { main }

{ Get the pathname and convert it to internal format using VFMT. }

{ Create the object. }
    ios_$create (pathname,
                 namelength,
                 records_$uid,          { Type UID    }
                 ios_$no_pre_exist_mode, { Error if exists }
                 [ios_$write_opt],      { Write access }
                 stream_id,
                 status);
    check_status;

{ Get record information.  }
    writeln ('Tye employee name or CTRL/Z to stop):');
    WHILE NOT eof DO
    BEGIN

        readln(info_rec.name);

        { Call internal procedure to calculate the namelength of
          employee name and put in namelen field. }
        put_name_length;

        writeln('Type employee id #:');
        readln(info_rec.emp_id);

        { Put employee ID field into the record. }

        ios_$put ( stream_id,                   { Stream ID of open object }
                   [ios_$partial_record_opt],   { Put options }
                   info_rec.emp_id,             { Data buffer }
                   SIZEOF(info_rec.emp_id,      { Length of data buffer }
                   status);
        check_status;

        { Write name field and terminate record. }
        buflen := info_rec.namelen;  { Record length varies with
                                       length of name field }

        ios_$put ( stream_id,      { Stream ID of open object }
                   [],             { Put options }
                   info_rec.name,  { Data buffer }
                   buflen,         { Length of data buffer }
                   status);
        check_status;

        writeln ('Type the next employee name or CTRL/Z to stop:');
    END;{while}
END. {  ios_partial_rec }
```

Example 4-10. Writing Variable-Length Records (Concluded)

4.10.3. Reading Fixed-Length Records with Seek Keys

Example 4-11 is a program that opens a stream to an object containing fixed-length records. This program reads the records sequentially, and then numbers them so it can later use IOS_$SEEK to seek to the record that the user specifies randomly.

This program performs the following:

- Declares a Pascal record containing the same fields as the program that created the record object, (in this case, ios_$put_rec_fixed.pas).
- Declares a seek key that corresponds with the record that you want to seek to.
- Reads the record-oriented object, and writes each record to the screen. It numbers each record beginning with zero (since records are zero-based).
- Asks the user to specify the number of the record to update, and assigns the number to "choice_rec." "Choice_rec" serves as the seek key for the record.
- Moves the stream marker to the requested record using IOS_$SEEK. This is an absolute seek because we want the offset to be calculated from the beginning of the object (which is record number 0). Since the user specifies the number of the desired record, it corresponds to the beginning of the object.

```
PROGRAM ios_seek_fixed_rec;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

TYPE  { Define the record type }
  info_rec_t = RECORD
      emp_id   : integer;
      address  : string;
      name     : string;
  END;

VAR
  status : status_$t;
  count  : integer;

  { $OPEN variables }
  pathname    : name_$pname_t;
  namelength  : integer;
  stream_id   : ios_$id_t;

  { $GET variables }
  line      : string;
  ret_len   : integer32;
  info_rec  : info_rec_t;
```

Example 4-11. Seeking Fixed-Length Records

```
  { $SEEK variable }
  choice_rec    : integer32; { Record number user wants changed }
                             { This serves as the seek key }
  no_of_recs    : integer;   { Number of records in object }
  response      : char;

BEGIN { main }

    { Get the pathname of a record-oriented object. }
    writeln;
    writeln ('Type pathname of a fixed-length record object to update: ');

    { Convert pathname to internal format using VFMT. }
    namelength := sizeof(pathname);     { Maximum namelength }

    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);

    { Open the object. }
    stream_id := ios_$open ( pathname,
                             namelength,
                             [ios_$write_opt],    { Write access }
                             status);
    check_status;

    no_of_recs := 0;          { Initialize to zero. }

{ Read and print the records and record numbers contained in the object }
{ until you read the entire object, or encounter an error. }

    WHILE status.all = status_$ok DO BEGIN

        ret_len := ios_$get ( stream_id,
                              [],               { Get options }
                              info_rec,
                              SIZEOF(info_rec),
                              status);

        IF (status.all = ios_$end_of_file) THEN
            EXIT
        ELSE
            check_status;
```

Example 4-11. Seeking Fixed-Length Records (Cont.)

```
    { Print and increment the record number.     }
    {  Note that record numbers are zero-based. }
    writeln;
    writeln('Record Number: ',no_of_recs:1);
    no_of_recs := no_of_recs + 1;

    { Print the employee ID, name and address. }

    writeln('Employee Number: ', info_rec.emp_id:1);
    writeln('Name: ', info_rec.name);
    writeln('Address: ', info_rec.address);
    writeln;

END;{WHILE}

{ Update the addresses. }

write( 'Type the number of the record you would like to update:');
writeln(' (0 - ', no_of_recs-1:1, ') or type CTRL/Z to stop:');

WHILE NOT eof DO BEGIN
    readln(choice_rec);

    { Test record choice }
    WHILE (choice_rec < 0) OR (choice_rec > no_of_recs) DO
    BEGIN
        write ('Record number is out of range. Enter a number');
        writeln(' between 0 and ', no_of_recs:1, ':');
        readln(choice_rec);
    END;

    { Move to the specified record -- using absolute record seek. }

    ios_$seek ( stream_id,
                ios_$absolute, { Seek_base }
                ios_$rec_seek, { Seek_type }
                choice_rec,    { Offset    }
                status);
    check_status;

    { Read the record. }
    ret_len := ios_$get ( stream_id,
                          [],               { Get options }
                          info_rec,
                          SIZEOF(info_rec),
                          status);
    check_status;

    { Print the employee ID, name and address. }
    writeln('Employee Number: ', info_rec.emp_id:1);
    writeln('Name: ', info_rec.name);
    writeln('Address: ', info_rec.address);
    writeln;
```

Example 4-11. Seeking Fixed-Length Records (Cont.)

```
        { Prompt for confirmation. }
        write('Would you like to update the address?');
        writeln(' (Y or N): ');
        readln(response);
        IF (response = 'Y') OR (response = 'y') THEN
        BEGIN
            writeln('Type the new address on one line: ');
            readln(info_rec.address);

            { Reposition stream marker to beginning of the record.}
            ios_$seek ( stream_id,
                        ios_$absolute, { Seek_base }
                        ios_$rec_seek, { Seek_type }
                        choice_rec,    { Offset    }
                        status);
            check_status;

            { Update the record. }
            ios_$put ( stream_id,
                       [],
                       info_rec,
                       SIZEOF(info_rec),
                       status);
            check_status;

            writeln('Record updated to contain the following: ');
            writeln('Address: ', info_rec.address);

        END;{if }

        { Prompt for next record to be updated. }
        writeln;
        write( 'Type the number of the record you would like to update:');
        writeln(' (0 - ', no_of_recs-1:1, ') or CTRL/Z to stop:');

    END; {while}
END. { ios_seek_fixed_rec }
```

Example 4-11. Seeking Fixed-Length Records (Concluded)

4.10.4. Record Formats

Usually, an application program using record-oriented objects need only know that a record-oriented object exists so that the program can perform I/O operations that recognize record boundaries. Users will rarely need to know how a type manager implements the record format.

However, should the need arise, you can inquire about a record's format using the IOS_$INQ_REC_TYPE call. You can change the record format or change the size of a fixed-length record with the IOS_$SET_REC_TYPE call.

Any type manager can implement some or all of the following record formats. The DOMAIN record-oriented type (REC) supports *most* of the following record formats. Another type manager may choose to implement a different subset. Because of this, some of the record types described in this section may not be applicable for your specific object type.

Table 4-15 lists of the various record formats with their predefined value.

Table 4-15. Available Record Formats

Predefined Value	Record Format
IOS_$F2	Fixed-length records
IOS_$V1	Variable-length records
IOS_$UNDEF	Unstructured records
IOS_$F1	Fixed-length records without a count field
IOS_$EXPLICIT_F2	Fixed-length records that *cannot* be changed to variable-length records

A fixed-length record object contains any number of records of the same length. A variable-length record object contains any number of records that vary in length.

In IOS_$F2, IOS_$V1, and IOS_$EXPLICIT_F2 formats, a record begins with a count field indicating the the number of bytes of data in the record. (Only the type manager ever reads or writes to a record's count field.)

Since fixed-length records have the same length, the count field at the beginning of each record in the object has the same value. Although this seems redundant, managers that implement IOS_$F2 typically maintain a count field so the object can be eventually handle variable-length records. For example, the DOMAIN REC type allows applications to change a fixed-length record object to variable-length records simply by writing records whose size differs.

An applications program can prevent the IOS manager from implicitly changing a fixed-length object to variable-length by specifying the IOS_$EXPLICIT_F2 record format. In this case, the type manager returns an error if a user tries to write variable-sized records to a fixed-record object.

A type manager can implement a fixed-length record format in a different manner. It can keep track of the size of the fixed-length records at the beginning of the object, rather than repeating the size of the record at the beginning of each record. In this case, the type manager uses the IOS_$F1 format.

Figure 4-1 illustrates how record-oriented objects with count fields are stored.

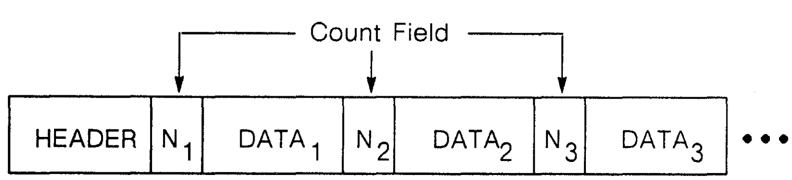


Figure 4-1. Record-Oriented Object with Count Fields

Figure 4-2 shows how a record object without a count field could be stored. (Just how it is stored depends on how the type manager implements it.)

Figure 4-2. Record-Oriented Object without Count Fields

Figure 4-3 shows how a record object without any structure could be stored. (Just how it is stored depends on how the type manager implements it.)

Figure 4-3. Unstructured Record-Oriented Object

Chapter 5
Using the Display Manager

The DOMAIN operating system has three components that affect the appearance of the display. You can use the following:

- The Display Manager to display text by manipulating pads and frames with PAD system calls. Use the Display Manager when you want to create windows, window panes, and manipulate text.

- The Graphics Primitives Resource (GPR) to perform graphics operations on DOMAIN displays. Use graphics primitives when you want to use graphics or mix graphics and text within windows and window panes.

- The black-and-white display driver (SMD) to gain more direct control over black-and-white displays. SMD calls do not work on color displays. You will rarely need to use this driver directly since both the Display Manager and the graphics primitives use this lower-level component. Also, there is a graphics primitive that corresponds to most SMD calls.

This chapter provides an overview of the Display Manager and describes how to use the system calls with the PAD prefix. It also describes calls to the paste buffer manager (PBUFS), which maintains buffer files; and calls to the touchpad manager (TPAD), which handles the touchpad and mouse. The graphics primitives are described in the *Programming with DOMAIN Graphics Primitives* manual. The SMD calls are described in *DOMAIN System Call Reference* manual.

DOMAIN has a separate graphics package, the DOMAIN 2D Graphics Metafile Resource (2D GMR) for graphics applications programming. For more information on the 2D GMR package, see the *Programming With DOMAIN 2-D Graphics Metafile Resources* manual.

5.1. System Calls, Insert Files, and Data Types

To manipulate the Display Manager, use system calls with the prefix PAD. In order to use PAD system calls, you must include the appropriate insert file in your program. The PAD insert files are:

/SYS/INS/PAD.INS.C	for C programs.
/SYS/INS/PAD.INS.FTN	for FORTRAN programs.
/SYS/INS/PAD.INS.PAS	for Pascal programs.

To use paste buffers within your program, use the system calls with the prefix PBUFS. You must also include the appropriate insert file. The PBUFS insert files are:

/SYS/INS/PBUFS.INS.C	for C programs.
/SYS/INS/PBUFS.INS.FTN	for FORTRAN programs.
/SYS/INS/PBUFS.INS.PAS	for Pascal programs.

To manipulate the touchpad or mouse in your program, use the system calls with the prefix TPAD. You must also include the appropriate insert file. The TPAD insert files are:

/SYS/INS/TPAD.INS.C	for C programs.
/SYS/INS/TPAD.INS.FTN	for FORTRAN programs.
/SYS/INS/TPAD.INS.PAS	for Pascal programs.

This chapter is intended to be a guide for performing certain programming tasks; the data and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference* manual.

5.2. Overview of the Display Manager

You use the Display Manager to manipulate the video display or screen, create and edit files, and monitor ongoing processes. By using PAD system calls, you can manipulate the appearance of the screen in many ways. This chapter describes how to

- Create windows and window panes through which the user can view part or all of a pad.
- Change window position and appearance, such as making them invisible, borderless, and having different character fonts.
- Create icons, change windows into icons, and change icon characters.
- Create and manipulate a frame to handle two-dimensional character I/O.
- Prevent user input from echoing on the screen with raw mode processing.

To start, we need to define a few terms used to describe the different components of your node's display. You are familiar with most of these terms already; this section merely summarizes them. For more information, see the *DOMAIN System User's Guide.*

Windows are the areas on the screen through which you view files and processes. With PAD calls, you can change a window's size and position on the screen and its position over the pad. Note that windows are not objects that any program recognizes; a program recognizes pads. Think of windows as the user's perspective. Most graphics applications refer to the Display Manager window as a **viewport.**

Pads are files that contain text and graphics. You can see material within a pad by looking through windows open into the pad. Note that the attributes that control the appearance and use of the text and graphics in a window are associated with the pad, not the window. Window attributes only control what parts of the pad are visible, and where on the screen they appear.

There are three types of pads: transcript, input, and edit pads.

Input pads accept keyboard input and transfer input to a program one line at a time. For example, the Shell input pad is the pad with the **$** prompt. Your programs can read from, but not write to, an input pad.

Transcript pads are associated with each input pad. The transcript pad contains a record (or transcript) of the program's dialogue with the user. That is, your program writes its output to the transcript pad after reading input from the input pad. Because it is a record, you can scroll the transcript pad backwards to view previous dialogue.

Edit pads are files that your program's users can edit, using the Display Manager. You can create edit window panes to let your program's users use the Display Manager's edit functions to format input to the program.

Read-only edit pads are edit files that the Display Manager opens for the users' viewing, but they cannot modify them. Note that once your program creates a read-only edit pad, it cannot be modified. Neither your program nor the keyboard user can execute a Display Manager command to turn a read-only edit pad into an edit pad, if your program created it as a read-only edit pad.

A line is the most common way to input information to a pad. Lines can contain text and a few control characters (such as TAB, BACKSPACE, and NEWLINE).

A frame is another way to write information to a pad. Instead of sending information line by line, a frame displays information from a two-dimensional area of any size. It can contain a broad range of text and graphic information. Within the frame, a program can move the cursor both horizontally and vertically, and write at any point. Frames are useful for simple graphics applications.

Window panes are separate areas of a window devoted to separate activities. Each pane acts as a window. The DOMAIN Language Level Debugger (DEBUG) is a good example of using window panes. When you invoke the debugger with the -SRC option, it runs in a window that is divided into five window panes, containing: the transcript of the debugging session, the debugger's input pad, input and transcript pads of the program you are debugging, and a copy of the program. Figure 5-1 shows the DEBUG display with the -SRC option.

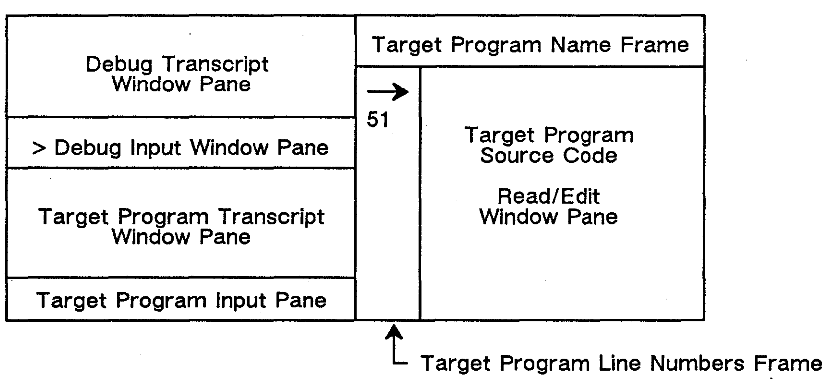


Figure 5-1. The DEBUG Display with the -SRC Option

The next few sections describe how you can use the Display Manager system calls to create and manipulate these pads.

5.3. Starting Out

Usually, you run most of your user programs in the user's Shell process, using the input and transcript pads already created by the Display Manager command, create process (CP). In most cases, these pads will suit your program's needs.

In some cases though, your program may need to create new pads, and windows or window panes to view them. You will want to create new pads when your program:

- Does not inherit any pads from the user. (When the program runs by a create process only (CPO) command, or by PGM_$INVOKE.)

- Needs to perform I/O in multiple contexts, or windows.

This section describes how to create a new transcript pad, and, if necessary, a window through which you can view it.

You can either create the transcript pad in a window pane and have your program run in the user's Shell window, or you can create a separate window and have your program run in its own window. Once you create the new transcript pad, you can create additional panes and frames to further subdivide the window.

Whether you create a pane or separate window depends mainly on your application, and its users. By creating your process within a window pane, you allow the user to have more control over the display itself. When your process runs within a window pane, it can create additional panes and frames within that pane. But it doesn't have anything to do with other areas of the user's display. This approach is often the best for experienced technical users. For example, programmers in a development environment often use multiple processes, and usually like to have control over the display.

If you create separate windows, your user has less control over the display, because your process chooses where to locate its windows on the display. This approach is useful when the user is mainly interested in the application. For example, in a process control application, users are usually interested in surveying the process statistics running in separate windows on the display; they don't want to change the display itself.

If your program creates windows, you should try to consider how much control you want to give to the users. For example, the DOMAIN system's alarm server creates windows in a controlled way -- it allows users to move the windows, and change their size. You can also use PAD calls to "remember" how the user set up the display, so you can position icons and windows according to where the user wants them.

5.3.1. Creating a New Pad in a New Window

To create a new transcript pad in a new window, use the PAD_$CREATE_WINDOW call. Example 5-1 shows how you can create a transcript pad, using PAD_$CREATE_WINDOW. An explanation of the arguments follows this example.

You can also create a window in icon format. It is the same as creating a full-sized window, but it first appears in icon format. For more information on icons, see Section 5.6.

```
{ Set the size and position of the future window. }

    window.top     := 300;
    window.left    := 300;
    window.width   := 300;
    window.height  := 300;

    pad_$create_window(' ',              { Null pathname for transcript pad. }
                       0,                { Null namelength for transcript pad. }
                       pad_$transcript,  { Type of pad. }
                       display_unit,     { No. of unit, usually 1. }
                       window,           { pad_$window_desc_t }
                       stream_win,       { stream_$id_t of the new window }
                       status);          { Completion status }
```

Example 5-1. Creating a New Pad with PAD_$CREATE_WINDOW

The arguments for **pathname** and **namelength** both have null values, because the transcript pad is normally a temporary pad that the Display Manager deletes when you close the pad.

The argument, **PAD_$TRANSCRIPT**, indicates that the pad created is a transcript pad. **Display_unit** indicates the unit number of the display on which the window will appear. This parameter is reserved for future use; you should always pass the value 1. **Window** indicates the position the new window will have on the display. You can set the window position by assigning values to window prior to the call.

Stream_win is the stream ID of the new window, in STREAM_$ID_T format, returned by this call. **Status** is the completion status returned by this call.

5.3.2. Creating a New Pad in a Window Pane

When you create a new transcript pad within a window pane, you associate your process with an existing window on the user's screen. To create the pad in a window pane, use the system call PAD_$CREATE. With PAD_$CREATE, you specify the stream to which you are relating this new window pane. Since you are associating your process with the user's standard output stream, you can either specify STREAM_$STDOUT or STREAM_$ERROUT.

Example 5-2 creates an original transcript pad from the user's standard output stream. An explanation of the arguments follow this figure.

```
    pad_$create ( ' ',              { Null pathname for transcript pad }
                  0,                { Null namelength for transcript pad }
                  pad_$transcript,  { Type of pad }
                  stream_$stdout,   { Relate to standard output stream pad }
                  pad_$left,        { Side of pad new pad will take up }
                  [],               { Size is relative to related pad }
                  100,              { New pad takes up 100 % of related pad }
                  stream_out,       { Stream ID of new pad }
                  status );         { Completion status }
```

Example 5-2. Creating a New Pad with PAD_$CREATE

The first two arguments indicate the **pathname** and **namelength**, respectively. As in PAD_$CREATE_WINDOW described above, you need not specify values if you are creating transcript pads. If you do not, they are temporary files, which go away when the stream closes.

PAD_$TRANSCRIPT indicates the type of window pane you are creating. You must specify PAD_$TRANSCRIPT when creating a transcript pad.

STREAM_$STDOUT is the stream ID, in STREAM_$ID_T format, of a pad to which this new pad is related. Since you want to relate your original transcript pad to the user's standard output, you can either specify STREAM_$STDOUT or STREAM_$ERROUT.

PAD_$LEFT indicates where the new window pane will be positioned, in relation to its related transcript pad. You can specify any one of the following positions:

- PAD_$LEFT for the left side of the transcript pad.
- PAD_$RIGHT for the right side of the transcript pad.
- PAD_$TOP for the top of the transcript pad.
- PAD_$BOTTOM for the bottom of the transcript pad.

An empty set of brackets, [ ], is the default pane_options attribute. The value of this argument determines, among other things, the interpretation of the next argument, pane_size. **Pane_size** specifies the height of the new window pane. When creating a new transcript pane, you must specify the default relative value (with empty brackets, [ ]).

Relative value means that the value of pane_size given in the next argument is relative to the size of its related window. In this case, the height of the new pad takes up the entire (100%) window.

Stream_out is the stream ID of the new window pane, in STREAM_$ID_T format, returned by this call. **Status** is the completion status returned by this call.

5.4. Creating Subsequent Pads in Window Panes

Once you have started your process on the user's display (either by associating your process with the user's pads, or creating your own pads, as described in Section 5.3), you can associate other pads, window panes and frames with it. This section describes how to create window panes. Section 5.7 describes how to create frames.

Most often, you will want to associate an input pad with your program's transcript pad. You might also want to divide your window into separate window panes, or you might want to create a frame to hold two-dimensional output.

You can have any number of window panes associated with the original transcript window, up to the Display Manager's limit of 40 pads and 60 windows. Just how many pads and panes you want depends on how many different kinds of output you want displayed concurrently.

You create subsequent pads and window panes within a window with the PAD_$CREATE system call, which we described in Section 5.3.2. You can create a pane of any one of the following types:

- PAD_$INPUT
- PAD_$EDIT
- $PAD_$READ_EDIT
- PAD_$TRANSCRIPT.

The following sections describe how to use PAD_$CREATE to create the three types of window panes.

5.4.1. Creating Input Pads in Window Panes

You will want to create an input pad to get input from the keyboard user. To create an input pad, use the PAD_$CREATE call, specifying **PAD_$INPUT** as the third argument. This call creates an input pad (and a window pane to view it), and associates it with a previously created transcript pad. (You must create a transcript pad *before* the associated input pad.)

```
NOTE:  You do NOT need to create an input pad if you are using
       the transcript pad for GPR direct mode graphics only.
```

You can have only one input pad for each transcript pad, and it must be located on the bottom of the pad. Example 5-3 shows how to create an input pad with PAD_$CREATE. An explanation of each argument follows the example.

```
pad_$create (' ',            { Null pathname for input pad }
             0,              { Null namelength for input pad }
             pad_$input,     { Type of pad }
             stream_out,     { Stream ID of related transcript pad }
             pad_$bottom,    { Input pads always go on bottom }
             [ ],            { Pane size is relative to transcript pad }
             20,             { New pad takes up 20% of related window }
             input_stream,   { Stream ID of this input pad }
             status );       { Completion status }
```

Example 5-3. Creating an Input Pad in a Window Pane

You must specify a null **pathname** and **namelength** when creating an input pad. **PAD_$INPUT** indicates that the type of window pane you are creating is an input pad.

Stream_out is the stream ID, in STREAM_$ID_T format, of a previously created transcript pad to which this pad is related. (In this case, the transcript pad is stream_out.)

PAD_$BOTTOM indicates that the new window pane will be positioned at the bottom of its related transcript pad. You must specify the bottom when creating an input pad. If you create additional transcript and edit window panes in a transcript window pane, the input window remains at the bottom of its associated transcript pane.

An empty set of brackets, [ ], indicates the default pane_options attribute. The value of this argument determines, among other things, the interpretation of the next argument, pane_size. **Pane_size** specifies the height of the new window pane. The value of pane_size is the maximum height the input window pane will ever be. All input pads start out to hold a single line of text in the current font. However, in cases where the user types input before the program is ready to read it, there may be more lines of input waiting for action. To accommodate this, you specify a larger window pane size for an input pad. A common value for the pane size is 20.

When PAD_$CREATE creates an input pad, it returns the stream ID of an input stream. Your program can read any keyboard input the user types into this pane. The Display Manager usually echoes the input into the related transcript pad. If you do not want the input to be echoed, you can specify the pane_options attribute [PAD_$INIT_RAW]. PAD_$INIT_RAW indicates that the input will be processed in raw mode, which prevents the system from preprocessing the input. Raw mode processing is described in the section below, 5.8.2.

Input_stream is the stream ID of the new window pane, in STREAM_$ID_T format, returned by this call. **Status** is the completion status returned by this call.

5.4.2. Creating Transcript Pads in Window Panes

You can associate other transcript window panes on top of the original transcript pad. To create a transcript pane, use the PAD_$CREATE call, specifying **PAD_$TRANSCRIPT** as the third argument.

Example 5-4 shows how to create a transcript pad with PAD_$CREATE. An explanation of each argument follows the example.

```
pad_$create ('transpathname', { Pathname }
             namelength,      { Namelength}
             pad_$transcript, { Type of pad }
             stream_out,      { Stream ID of related transcript pad }
             pad_$right,      { Side of original pad that new pad is located }
             [pad_$abs_size], { Pane size is absolute value }
             30,              { New pad is 30 lines high (scaled) }
             trans_stream,    { Stream ID of this transcript pad }
             status );        { Completion status }
```

Example 5-4. Creating a Transcript Pad in a Window Pane

You can specify either null, or a **pathname** and **namelength** when creating a transcript pad and pane. If you specify null, the transcript pad is a temporary file, which goes away when the program ends.

If you specify the pathname of an existing file for a transcript pad, the Display Manager positions the pad at the beginning of the file, but scrolls down to the bottom of the file the first time the user writes to the pad. Creating a transcript window pane whose pad is an existing file is a convenient way for your program to display prepared text or graphics, such as menus. The Display Manager can call an existing file to the screen faster than your program can create it.

If you create a transcript window pane with a pathname that does not refer to an existing pad, the Display Manager creates a new permanent file. Thus, the program dialogue is a permanent record that you can refer to after the program terminates.

PAD_$RIGHT indicates that the new pad will be at the right side of the associated pad. You can place the transcript pad anywhere on the original transcript pad, so you can specify any of the following options: PAD_$TOP, PAD_$BOTTOM, PAD_$RIGHT or PAD_$LEFT.

PAD_$ABS_SIZE indicates that the next argument, pane_size, will be an absolute value, according to the current scale factor. That is, pane_size will be 30 lines high in the current font, if the scale factors are set to the default, 0,0. For details on scale factors, see Section 5.5.7. By specifying an absolute size, the Display Manager attempts to keep the pane at that size, even if its related window grows or shrinks. However, the window pane can never be larger than its related window, so that if the window shrinks below the size of the window pane, the window pane must also shrink. You can also specify the default relative value with empty brackets, [ ]. This makes the new pad's size a percentage of the original pad.

Trans_stream is the stream ID of the new window pane, in STREAM_$ID_T format, returned by this call. **Status** is the completion status returned by this call.

5.4.3. Creating Edit Pads in Window Panes

An edit window pane is a window pane where the user can type or edit text with the usual Display Manager text-editing commands. If your program requires a large amount of input from them, you can create an edit window pane for users to enter their data.

To create an edit pad, use the PAD_$CREATE call, specifying **PAD_$EDIT** as the third argument. This call creates an edit pad (and a window pane to view it) and associates it with a previously created transcript pad.

Example 5-5 shows how to create an edit pad with PAD_$CREATE. An explanation of each argument follows the example.

```
pad_$create ('editpathname',  { Pathname }
             namelength,      { Namelength}
             pad_$edit,       { Type of pad }
             stream_out,      { Stream ID of related transcript pad }
             pad_$top,        { Side of original pad that new pad is located }
             [pad_$abs_size], { Pane size is absolute value }
             30,              { New pad is 30 lines high ( scaled ) }
             edit_stream,     { Stream ID of this transcript pad }
             status );        { Completion status }
```

Example 5-5. Creating an Edit Pad in a Window Pane

You can specify a **pathname** and **namelength** when creating an edit pad. If you give a pathname of an existing file, the user sees and can edit that file. If you give a new pathname, the user's input goes into a new, permanent file. If you supply no pathname for the edit file, the user's input goes away when the stream closes.

PAD_$TOP indicates that the edit pad is located at the top of the associated pad. You can place the edit pad anywhere on the pad, and can specify any of the following sides: PAD_$TOP, PAD_$BOTTOM, PAD_$RIGHT, PAD_$LEFT.

PAD_$ABS_SIZE indicates that the next argument, pane_size, will be an absolute value, according to the current scale factor. (That is, pane_size will be 30 lines high.) By specifying an absolute size, the Display Manager attempts to keep the pane at that size, even if its related window grows or shrinks. However, the window pane can never be larger than its related window, so that if the window shrinks below the size of the window pane, the window pane must also shrink. You can also specify the default relative value with empty brackets, [ ]. This makes the new pad's size a percentage of the original pad.

Edit_stream is the stream ID of the new window pane, in STREAM_$ID_T format, returned by this call. **Status** is the completion status returned by this call.

After you create an edit window pane, you can then call PAD_$EDIT_WAIT. This suspends the process until the user terminates the edit session in the edit pane with a CRTL/Y, CTRL/N, EXIT, or ABORT (WC or WC -Q) command. The process then gains control, closes the window, thereby allowing your program to access the information.

After an editing session, the program has different access privileges to the edited file depending on when the file was created. If the file is a temporary file, specified by a null pathname in PAD_$CREATE, the program has read and write access to it. However, if the file is a pre-existing file, specified as the pathname in PAD_$CREATE, your program has only read access to it. You can change the file access, if necessary by using the STREAM_$REDEFINE system call described in Chapter 4.

5.4.4. Creating Read-Only Edit Pads in Window Panes

A read-only edit pad is a file that users can read but not modify. To create a read-only edit pad, use the PAD_$CREATE call, specifying **PAD_$READ_EDIT** as the third argument. This call creates a read-only edit pad (and a window pane to view it) and associates it with a previously created transcript pad. For a description of the call, see Section 5.4.3, Creating Edit Window Panes.

Note that once you create an edit pad as read-only, the user *cannot* change it into an edit pad. The Display Manager command that turns a read-only edit pad into an edit pad does not work when the window pane is created with PAD_$CREATE. A read-only edit pad must refer to an existing file.

5.4.5. Closing Windows and Window Panes

A pad closes when its associated stream closes; the stream closes when your program makes a STREAM_$CLOSE system call, or when your program terminates, regardless of whether the termination is normal or unexpected. It is good practice to use the STREAM_$CLOSE system call to close any opened I/O streams before you conclude your program.

You should close the streams in the reverse order that you created them, so that you close the original transcript pad last. You can close an edit pad stream while the user is still editing. This denies your program further access to the file, but allows the user to finish editing it. Even though a pad closes when your program ends, some types of windows or window panes associated with these pads do *not* close automatically when their associated streams close. These include

- Transcript windows (not panes).
- Edit windows and panes.
- Read/edit windows (not panes).

If you want these windows or panes to close when their related streams close, use the PAD_$SET_AUTO_CLOSE call. Usually, you would include this call soon after you create the window or window pane in case your program terminates unexpectedly. (A user can type the Display Manager WC -A command to achieve the same results as a PAD_$SET_AUTO_CLOSE.)

You do not need to use PAD_$SET_AUTO_CLOSE with input pads, transcript panes, or read/edit panes, because they go away automatically when their associated streams close.

5.4.6. Sample Program: Creating and Closing Windows and Window Panes

Example 5-6 is a program that shows how to use PAD calls to create an original transcript pad and subsequent window panes. It also shows how to use the PAD_$SET_AUTO_CLOSE and STREAM_$CLOSE system calls.

```
PROGRAM pad_make_windows;

{ This program makes a new transcript pad and window, and associates
   other window panes. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

CONST
  display_unit    = 1;
  window_count    = 1;
  auto_close      = TRUE;

VAR
  source_stream   : stream_$id_t;
  input_stream    : stream_$id_t;
  edit_stream     : stream_$id_t;
  seek_key        : stream_$sk_t;

  window          : pad_$window_desc_t;
  window_list     : pad_$window_list_t;
  window_size     : integer;

  status          : status_$t;
  pathname        : name_$pname_t;
  namelength      : integer;
  count           : integer;
```

Example 5-6. Creating and Closing Windows and Window Panes

```
{* ************************************************************************ *}
{* Procedure CHECK_STATUS to check for errors.  It prints an error message,   *}
{* and exits on bad status.                                                   *}
{* ************************************************************************ *}

PROCEDURE check_status;

BEGIN
    IF status.all <> status_$ok THEN BEGIN
        error_$print( status);
        pgm_$exit;
    END;
END;  { check_status}

{* ************************************************************************ *}
{* Procedure HOLD_DISPLAY to suspend program to demonstrate how calls work.   *}
{* ************************************************************************ *}

{ This internal procedure calls TIME_$WAIT to suspend the process for 3 seconds
  so you can see how each call works.  }

PROCEDURE hold_display;

VAR
  rel_time      : time_$clock_t;

BEGIN { hold_display}
    cal_$sec_to_clock ( 3, rel_time );  { Convert secs to UTC value }
    time_$wait ( time_$relative,
                 rel_time,
                 status );              { Time to wait }
    check_status;
END;  { hold_display }

{* ************************************************************************ *}

BEGIN  { Main }

    { Set position of future window.  }

    window.top    := 150;
    window.left   := 150;
    window.width  := 450;
    window.height := 450;

    { Create original transcript pad and window. }

    pad_$create_window( ' ',              { No pathname for transcript pad }
                        0,                { No namelength for transcript pad }
                        pad_$transcript,  { Type of pad }
                        display_unit,     { Number of display unit }
                        window,           { Position of window }
                        source_stream,    { Returns stream ID }
                        status);          { Completion status }
    check_status;
```

Example 5-6. Creating and Closing Windows and Window Panes (Cont.)

```
{ Close window when stream closes. }

pad_$set_auto_close ( source_stream,  { Stream ID }
                      window_count,   { Number of window }
                      auto_close,     { Flag -- set to TRUE }
                      status);        { Completion status }
check_status;

{ Make an input pane at the bottom of the window. }

pad_$create ( ' ',           { Null pathname for input window }
              0,             { Null namelength }
              pad_$input,    { Type of pad }
              source_stream, { Same stream ID as window }
              pad_$bottom,   { New pane position on original pad }
              [],            { Pane height relative to original pad }
              20,            { Height maximum is 20% of original pad }
              input_stream,  { Returns stream ID of window pane }
              status );      { Completion status }
check_status;

{ Get pathname from keyboard and set values of pathname, namelength. }

WRITELN ('Type in the pathname of the file: ');

vfmt_$read2('%""%eka%.',
            count,
            status,
            pathname,
            namelength);
check_status;

{ Make an edit pane for the rest of the window above the input pad and
  associate it with specified file. }

pad_$create ( pathname,
              namelength,
              pad_$edit,
              source_stream, { Same stream ID as window }
              pad_$top,      { New pane position on original pad }
              [],            { Pane height relative to original pad }
              60,            { Height = 60% of pad minus input pad }
              edit_stream,   { Returns stream ID of window pane }
              status );
check_status;

{ Close edit pad when stream closes. }

pad_$set_auto_close ( edit_stream,    { Stream ID }
                      window_count,   { Number of window }
                      auto_close,     { Boolean -- set to TRUE }
                      status);        { Completion status }
check_status;
hold_display;
```

Example 5-6. Creating and Closing Windows and Window Panes (Cont.)

```
    { Close the streams. }

    stream_$close( edit_stream, status );
    check_status;

    stream_$close( input_stream, status );
    check_status;

    stream_$close( source_stream, status );
    check_status;

END.    { pad_make_windows }
```

Example 5-6. Creating and Closing Windows and Window Panes (Cont.)

5.5. Manipulating Windows

There are many Display Manager calls that tell you about the display, and allow you to change it. For example, if you run your process in the user's Shell process, you can use various Display Manager calls to find out about the display.

The following sections describe how to inquire about window positions and change them, pop windows to the foreground of the screen, push them to the background, make them invisible, re-appear, and borderless. It also describes how to change character fonts and scale factors.

5.5.1. Specifying a Window Number with PAD_$INQ_WINDOWS

Most of the PAD calls that manipulate windows require that you specify the stream ID and number of the desired window. You must specify a window number because a user might have more than one window viewing the same pad. This occurs any time a user and a program or two programs make a window on the same object. Typically, this can happen when the program calls PAD_$CREATE on an edit window that the user already has open on the display. Your program opens a second window, so it must refer to the number, 2, when it manipulates that window.

Assuming that you want to change the most current window viewing the pad, call PAD_$INQ_WINDOWS first. Since PAD_$INQ_WINDOWS returns the number of windows open to the pad, the number equals the latest window viewing the pad. In subsequent calls requiring a specific window number, use the number returned by PAD_$INQ_WINDOWS.

5.5.2. Getting Window Positions with PAD_$INQ_WINDOWS

PAD_$INQ_WINDOWS also tells you the size and position of each window viewing the pad. This is useful, for example, if your program display depends on whether the user's window is vertical or horizontal in shape, or if it needs to scale its output to fit in the window.

PAD_$INQ_WINDOWS returns the position of the window viewed to the pad in the order of top, left, width, and height, excluding the window's border and legend. If more than one window is open to the pad, you can get information about any number of windows.

Note that the values of top and left are expressed in raster units, but width and height are divided by the current scale factors. If you need to know the width and height in raster units, you can convert them using the system call PAD_$SET_SCALE prior to using PAD_$INQ_WINDOWS. Example 5-7 shows how to convert the width and height to raster units by using the call PAD_$SET_SCALE. The call changes the value of width and height to raster units when you specify x and y factors to be 1. See Section 5.5.7 for details on PAD_$SET_SCALE.

```
PROGRAM pad_inq_window_size;

{This program gets information about size of windows open to pad. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';

CONST
  max_windows     = 10;
  font_size       = 0;        { No need for font pathname }

VAR
  window_info     : pad_$window_list_t;
  n_windows       : integer;
  width_scale     : integer;
  height_scale    : integer;
  font_name       : pad_$string_t;
  font_len        : integer;
  bottom, right   : integer;
  status          : status_$t;
  i               : integer;

{* ********************************************************************** *}
{* Procedure Check_status for error handling.          (See Example 5-6).  *}
{* ********************************************************************** *}

BEGIN { Main Program }

    { Set scale to 1,1 to get width and height in raster units. }

    pad_$set_scale ( stream_$stdout,  { Standard output (display) }
                     1,               { x factor in raster units }
                     1,               { y factor in raster units }
                     status );        { Completion status }
    check_status;
```

Example 5-7. Getting Size and Position of Windows

```
    { Get window information about user's standard output stream. }

    pad_$inq_windows (stream_$stdout,  { Standard output (display) }
                      window_info,     { Current position of window }
                      max_windows,     { Maximum no. of windows desired }
                      n_windows,       { Number of windows open to pad }
                      status);         { Completion status }
    check_status;

    { Write window information to screen. }

    writeln;
    writeln ( '  ========================================= ' );
    writeln;

    IF (n_windows = 1) THEN
        writeln (' One window is open to this pad.')
        ELSE writeln (' There are ',  n_windows:1,
                      ' windows are open to this pad.');
    writeln;

    { Write window information for each window open to current pad. }

    FOR i := 1 to n_windows DO
        WITH window_info[i] DO

        BEGIN

             bottom := top + height;
             right  := left + width;

             { Write positions to display: }

             writeln (' Window ',i:1);
             writeln ('----------');
             writeln;
             writeln (' Upper left corner is at position (',
                        left:1,',',top:1,') ');
             writeln (' Lower right corner is at  position (',
                        right:1,',',bottom:1,')');
             writeln (' Width of window = ',width:1,
                        ' (raster units)');
             writeln (' Height of window  = ',height:1,
                        ' (raster units)');
             writeln ;

        END; {with}

        writeln ( '  ========================================= ' );

END.   { pad_inq_window_size }
```

Example 5-7. Getting Size and Position of Windows (Cont.)

5.5.3. Getting Position of Window Borders with PAD_$INQ_FULL_WINDOW

While PAD_$INQ_WINDOWS returns information about the screen space available to your program, PAD_$INQ_FULL_WINDOW returns information about an entire window in relation to the user's display. PAD_$INQ_FULL_WINDOW returns information that tells you how much of the display a window uses -- including its legend and border. Even if you specify a window pane, PAD_$INQ_FULL_WINDOW returns information about the outermost window related to the specified window pane. You might use this information if you want to position a window on the user's display so that it will not overlap an existing window. To do so, use PAD_$INQ_FULL_WINDOW to get the dimensions of the existing windows to calculate where to make the new window.

You can also use PAD_$INQ_FULL_WINDOW in programs that want to remember where the user last placed a window. Use PAD_$INQ_FULL_WINDOW to find out where the user positions the window, and then, if the window is recreated at some future time, you can call PAD_$SET_FULL_WINDOW to position the window in the same place. You can also use PAD_$SET_FULL_WINDOW to grow and move full windows.

Due to a current implementation restriction, if you use PAD_$SET_FULL_WINDOW on an invisible window, the call makes the window visible. You will have to use another PAD_$MAKE_INVISIBLE to make the window invisible again. Example 5-8 is an example of setting the position of a full window.

```
PROGRAM pad_full_window_show;

{ This program uses PAD calls to manipulate full windows. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';

CONST
  no_border    = FALSE;

VAR
  stream_one  : stream_$id_t;
  status      : status_$t;
  window      : pad_$window_desc_t;
  windowlist  : pad_$window_list_t;
  winlistsize : integer;
  window_no   : integer;
  full_window : pad_$window_desc_t;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.           (See Example 5-6).  *}
{* ************************************************************************ *}
{* Procedure Hold_display to demonstrate calls.         (See Example 5-6).  *}
{* ************************************************************************ *}
```

Example 5-8. Using PAD Calls to Manipulate a Full Window

```
BEGIN { Main Program }
    .
    . { Set original position of windows.
    .   Create a window with pad_$create_window. }

    pad_$inq_windows ( stream_one,      { Stream ID }
                       windowlist,      { Array of windows }
                       winlistsize,     { Number of windows to get info }
                       window_no,       { Returns number of windows }
                       status );        { Completion status }
    check_status;

    pad_$make_invisible( stream_one,
                         window_no,  { Returned by PAD_$INQ_WINDOWS }
                         status );
    check_status;

    pad_$inq_full_window ( stream_one,
                           window_no,
                           full_window, { Returns full window position }
                           status );
    check_status;

    pad_$set_full_window ( stream_one,
                           window_no,
                           full_window,
                           status );
    check_status;
    hold_display;

    pad_$make_invisible ( stream_one,
                          window_no,
                          status );
    check_status;

END.  { pad_full_window_show }
```

Example 5-8. Using PAD Calls to Manipulate a Full Window (Cont.)

5.5.4. Changing How Windows Look

You call PAD_$MAKE_INVISIBLE to make the specified window disappear; you call PAD_$SELECT_WINDOW to make it re-appear. The PAD_$POP_PUSH_WINDOW and PAD_$SET_BORDER calls use Boolean arguments to allow you to change the window appearance. For example, if the program sets the Boolean argument in PAD_$SET_BORDER to FALSE, PAD_$SET_BORDER removes the border from a window. If it is TRUE, PAD_$SET_BORDER adds the border. (By default, all windows have borders; PAD_$SET_BORDER adds the border only to windows made borderless by a previous call to PAD_$SET_BORDER.)

Example 5-9 is a sample program using these calls. Note that PAD_$SET_BORDER works only with full windows. You cannot create a borderless window pane or frame. If you create a borderless window and associate a window pane with that window, the border re-appears.

```
PROGRAM pad_window_show;

{ This program shows how to pop and push windows, make a
  window visible and invisible, and remove a window border. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

CONST
   display_unit  = 1;
   auto_close    = TRUE;
   no_border     = FALSE;
   pop           = TRUE;
   push          = FALSE;

VAR
    stream_one   : stream_$id_t;
    stream_two   : stream_$id_t;
    stream_three : stream_$id_t;
    pane_stream  : stream_$id_t;

    window_one   : pad_$window_desc_t;
    window_two   : pad_$window_desc_t;
    window_three : pad_$window_desc_t;

    window_no1   : integer;
    window_no2   : integer;
    window_no3   : integer;
    window_list  : pad_$window_list_t;
    status       : status_$t;

{* ************************************************************************** *}
{* Procedure Check_status for error handling.             (See Example 5-6).   *}
{* ************************************************************************** *}

BEGIN { Main Program }

    . { Set the original positions of the windows. }
    . { Create 3 transcript pads with full windows using pad_$create_window. }
    . { Make windows close when stream closes using pad_$set_auto_close. }
    . { Get value of window_no1, window_no2, and window_no3 for next calls }
    . { using pad_$inq_windows. }

    { Remove border from the last window. }

    pad_$set_border ( stream_three,     { Stream ID }
                      window_no3,       { Window number }
                      no_border,        { Set no border }
                      status);          { Completion status }
    check_status;
```

Example 5-9. Changing How a Window Looks

```
    { Push the last window open to the bottom. }

    pad_$pop_push_window ( stream_three,
                           window_no3,
                           push,          { Push window }
                           status);
    check_status;

    { Pop the last window open to the top. }

    pad_$pop_push_window ( stream_three,
                            window_no3,
                            pop,          { Pop window }
                            status);
    check_status;

    { Make the second window invisible. }

    pad_$make_invisible ( stream_two,        { Stream ID }
                          window_no2,        { Window number }
                          status);           { Completion status }
    check_status;

    { Make the first window invisible. }

    pad_$make_invisible ( stream_one,
                          window_no1,
                          status);
    check_status;

    { Make the first window visible again. }

    pad_$select_window  ( stream_one,         { Stream ID }
                          window_no1,         { Window number }
                          status);            { Completion status }
    check_status;

    { Create pad and window pane on borderless window, note that
      in doing so, the border re-appears. }

    pad_$create ( ' ',                   { Null pathname }
                  0,                     { Null namelength }
                  pad_$input,            { Type of pad }
                  stream_three,          { Stream ID of related pad }
                  pad_$bottom,           { Location on pad }
                  [],                    { Relative size }
                  20,                    { Height of pane (scaled ) }
                  pane_stream,           { Stream ID }
                  status);
    check_status;

    { Close streams before terminating program using stream_$close. }

END.   { pad_window_show }
```

Example 5-9. Changing How a Window Looks (Cont.)

5.5.5. Inquiring About the User's Display and Keyboard

You can use the system PAD_$INQ_DISP_TYPE call to find out about the user's display, and tailor your program's action according to it. For example, you can set up the position of your windows according to the type of display in use. Example 5-10 checks for the user's type of display.

NOTE: If you are using graphics through GPR or GMR, it is better to use the several GPR inquire calls to determine the specific display attributes (such as the x and y dimensions). This way, your program will be less device-dependent, and will continue to work when new display types are introduced.

You can use the system call PAD_$INQ_KBD to find out about a user's keyboard. For example, you might want to set up program definition keys according to the type of keyboard in use. Example 5-10 checks for the keyboard in use, and responds accordingly.

```
PROGRAM pad_inq_disp_kbd;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  max_windows    = 10;
  font_size      = 0;
  buffer         = 256;

VAR
  status         : status_$t;
  display_type   : pad_$display_type_t;
  unit_number    : integer;
  kbd_suffix     : pad_$string_t;
  suffix_length  : integer;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.            (See Example 5-6).   *}
{* ************************************************************************ *}
{* Procedure Hold_display to demonstrate calls.          (See Example 5-6).   *}
{* ************************************************************************ *}

BEGIN { Main Program }

    { Find out which type of display is in use. }
    pad_$inq_disp_type ( stream_$stdout, { Standard output stream - display }
                         display_type,   { Returns type of display }
                         unit_number,    { Returns unit number, always 1 }
                         status);        { Completion status }
    check_status;
```

Example 5-10. Inquiring About User's Display and Keyboard

```
writeln;
writeln ( '  ============================================ ');
writeln ( '    Number of display units: ');

IF unit_number = 1 THEN
    writeln ( ' There is one display unit connected to this node. ')
    ELSE
        BEGIN
            writeln ( ' There are ',unit_number,' display units') ;
            writeln ( ' connected to this node. ');
        END;
writeln;
writeln ( '  ============================================ ');
writeln ( ' Type of display: ');

CASE display_type OF

     pad_$bw_15p : writeln ( ' This is a black-and-white portrait. ');
     pad_$bw_191 : writeln ( ' This is a black-and-white landscape. ');
     pad_$color_display : writeln
                  ( ' This is a color display (1024 x 1024 pixels). ');
     pad_$800_color : writeln
                  ( ' This is a color display ( 1024 x 800 pixels). ');
     pad_$none : writeln ( ' There is no display. ');

END; { case }

{  Find out which keyboard is in use. }

pad_$inq_kbd ( stream_$stdout,    { Standard output stream }
               buffer,            { Size of string buffer }
               kbd_suffix,        { Returns keyboard suffix string }
               suffix_length,     { Returns keyboard suffix length }
               status );          { Completion status }
check_status;

writeln ( '  ============================================ ');

IF suffix_length = 0
    THEN BEGIN

        writeln ( ' The keyboard suffix is 0 ');
        writeln  ( ' User has the 880 keyboard. ');
        END
    ELSE IF kbd_suffix[suffix_length] = '2'
        THEN BEGIN
            vfmt_$write2 ( ' The keyboard suffix is:"%A" %. ',
                                  kbd_suffix, suffix_length );
            writeln ( ' User has the low-profile keyboard. ');
            END
        ELSE writeln  ( ' Not sure which keyboard is in use. ');
```

Example 5-10. Inquiring About User's Display and Keyboard (Cont.)

```
    { Redefine the keyboard function keys. }
    IF (suffix_length = 0) OR (kbd_suffix[suffix_length] = '2')
        THEN BEGIN

            writeln;
            writeln ( ' Redefining low-profile function keys. ');

            pad_$def_pfk ( stream_$stdout,    { Stream ID }
                           'F1',       { Keyname }
                           'TT',       { DM command -- to top of window }
                           2,          { Length of DM command }
                           status );
            check_status;
            hold_display;

            END;

END. { pad_inq_disp_kbd }
```

Example 5-10. Inquiring About User's Display and Keyboard (Cont.)

5.5.6. Specifying Character Fonts

You can specify different styles of character fonts that your program uses by changing the font file. A **font file** contains binary data that defines the size and shape of each character. Different font files define different typefaces (such as Times Roman or Old English), fonts (such as boldface or italic), and size (such as 5x9 or 7x13).

Traditionally, a typeface has various attributes such as size and font. However, the Display Manager font files do not make these distinctions. Instead, any variations of a typeface, font or size constitutes a different font file, and no relationships exist between font files.

Most font files reside in the directory /SYS/DM/FONTS. You can tell the type of font by its name. Fixed-width fonts begin with the letter **f**, and contain the size of the font in raster units. Some have an extension indicating that the file is a variant of a standard file. For example, "f5x9.b" is the boldface version of "f5x9". The extenstion, ".i" is the italics version, and ".iv" is the inverted (reverse-video) version of the font.

You can specify any one of the fonts listed in that file in your program. A pad can use up to 100 fonts at the same time. Before you use a font, you must call PAD_$LOAD_FONT to inform the Display Manager that you intend to use this font at some future time. Then you call PAD_$USE_FONT to specify the current font for your program to use. You can use PAD_$USE_FONT to switch between loaded fonts as often you want. The Display Manager displays a character in a window by copying the character's image from the current font to a specified location in the window.

For more information about font files, see the description of the font editor, EDFONT in the *DOMAIN System Command Reference* manual. This manual also lists the fonts available in /SYS/DM/FONTS in the section describing the font load (FL) command.

Example 5-11 shows how to use PAD_$LOAD_FONT and PAD_$USE_FONT to specify a font file. For another example of using fonts, see Example 5-20.

```
PROGRAM pad_font;

{ This program loads and uses fonts. It creates a transcript window, and
  writes out a message, using the inverted font, f5x9.iv.  The user
  can put the keyboard cursor inside the pad to see the message. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit   = 1;
  window_no      = 1;
  auto_close     = TRUE;
  trans_message  = ' This is a transcript pad.';
  new_font_name  = 'f5x9.iv';         { Fixed width inverted font }

VAR
  source_stream : stream_$id_t;
  pane_stream   : stream_$id_t;
  seek_key      : stream_$sk_t;
  status        : status_$t;
  window        : pad_$window_desc_t;
  new_font_id   : integer;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.             (See Example 5-6).   *}
{* ************************************************************************ *}
{* Procedure Hold_display to demonstrate calls.           (See Example 5-6).   *}
{* ************************************************************************ *}

BEGIN { Main Program }

    . { Set position of future window.  }
    . { Create original transcript pad and window. }
    . { Close window when stream closes. }

    { Load the standard inverted font, f5x9.iv, for transcript window. }

    PAD_$LOAD_FONT ( source_stream,          { Stream ID }
                     new_font_name,          { Font_name f5x9.iv }
                     SIZEOF(new_font_name),  { Length of font_name }
                     new_font_id,            { Returns font ID }
                     status);                { Completion status }
    check_status;
```

Example 5-11. Selecting a Character Font File

```
{ Use PAD_$USE_FONT to have program use the desired font. }

PAD_$USE_FONT ( source_stream,        { Stream ID }
                new_font_id,          { Font ID loaded above }
                status );
check_status;

{ Write name of file in transcript pad. }

vfmt_$ws2 ( source_stream,
            '%A%.',          { Add newline after string using VFMT }
            trans_message,
            SIZEOF(trans_message));
check_status;
hold_display;
   .
   .
   .
```

Example 5-11. Selecting a Character File File (Cont.)

5.5.7. Changing Scale Factors

Most system calls deal with screen locations by using absolute pixel (raster unit) coordinates. Some PAD calls require the size of the current font to describe the location of text in terms of lines and characters, rather than absolute locations or sizes.

These calls are:

- PAD_$CPR_ENABLE
- PAD_$CREATE (with the PAD_$ABS_SIZE option)
- PAD_$CREATE_FRAME
- PAD_$INQ_POSITION
- PAD_$LOCATE
- PAD_$MOVE
- PAD_$INQ_WINDOWS

For example, if you specify a five as the horizontal size in a PAD_$MOVE call, you do not mean five pixel locations, but rather five times the horizontal scale factor.

By default, the scale factor depends on the size of the font currently in use. You can change the scale factors to be in raster units by using the PAD_$SET_SCALE call. Normally, you specify one for x and y when you use PAD_$SET_SCALE, meaning the values of x and y will be in pixels rather than lines and columns. Note that a column starts at one, so when scale factors are according to lines and columns, the edge of the window is at column one. However, when scale factors are in raster units, the edge of the window is zero.

To restore the default font-size scaling, use PAD_$SET_SCALE, specifying zero as the value of the x and y scale factors.

Example 5-12 shows the difference between a pad created with the default scale factor, and a pad created after setting the scale to raster units with PAD_$SET_SCALE. Note that scaling factors are in effect because it specifies the PAD_$ABS_SIZE option when creating this pad.

```
PROGRAM pad_scale;

{ This program is a sample of using PAD_$SET_SCALE. The first
  window creates a transcript pad that is 5 lines high.  The second
  window creates a transcript pad that is 20 raster units high. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit = 1;
  auto_close   = TRUE;
  window_no    = 1;

VAR
  seek_key          : stream_$sk_t;
  stream_one        : stream_$id_t;
  stream_four       : stream_$id_t;
  pane_stream_one   : stream_$id_t;
  pane_stream_four  : stream_$id_t;
  status            : status_$t;
  window_one        : pad_$window_desc_t;
  window_two        : pad_$window_desc_t;

{* ************************************************************************* *}
{* Procedure Check_status for error handling.            (See Example 5-6).   *}
{* ************************************************************************* *}
{* Procedure Hold_display to demonstrate calls.          (See Example 5-6).   *}
{* ************************************************************************* *}

BEGIN { Main Program }

    . { Set position of future windows. }
    . { Open the window as a transcript pad. }
    .
    pad_$create_window ( ' ',
                         0,
                         pad_$transcript,
                         display_unit,
                         window_one,
                         stream_one,
                         status );
    check_status;
```

Example 5-12. Setting Scale Factors to Raster Units with PAD_$SET_SCALE

```
pad_$create ( ' ',
              0,
              pad_$transcript,
              stream_one,
              pad_$top,
              [pad_$abs_size],   { Pad is absolute value }
              5,                 { 5 lines high }
              pane_stream_one,
              status );
check_status;

{ Open the window as a transcript pad. }

pad_$create_window ( ' ',
                     0,
                     pad_$transcript,
                     display_unit,
                     window_two,
                     stream_two,
                     status );
check_status;

{ Set scale of window height and width to be in raster units. }

pad_$set_scale ( stream_four,
                 1,              { Scale factor for x-coordinate }
                 1,              { Scale factor for y-coordinate }
                 status );
check_status;

pad_$create ( ' ',
              0,
              pad_$transcript,
              stream_two,
              pad_$top,
              [pad_$abs_size],    { Pad absolute size }
              20,                 { Raster units }
              pane_stream_two,
              status);
check_status;
    .
    .
    .
```

Example 5-12. Setting Scale Factors to Raster Units (Cont.)

5.5.8. Getting Current Scale Factors with PAD_$INQ_FONT

If you set the scale factor to raster units, you might want to know the scale factor of the current font for another call. To do so, use PAD_$INQ_FONT. Example 5-13 sets the scale to raster units before creating a frame. To put the output cursor in the frame, it uses PAD_$MOVE. In PAD_$MOVE, the x and y coordinates indicate where to locate the character on the display. The y coordinate must be large enough to handle the height of the character font. To find out the height, it uses a call to PAD_$INQ_FONT. For details on frames, see Section 5.7.

```
PROGRAM pad_inq_font;

{ This program creates a frame at the top of the user's standard output pad,
  and writes the prompt "#" inside the frame.  }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit = 1;
  auto_close   = TRUE;
  prompt_str   = ('# ');
  max_windows  = 1;

VAR
  seek_key      : stream_$sk_t;
  pane_stream   : stream_$id_t;
  status        : status_$t;
  window_info   : pad_$window_list_t;
  n_windows     : integer;
  font_len      : integer;
  font_height   : integer;
  font_width    : integer;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.              (See Example 5-6).   *}
{* ************************************************************************ *}
{* Procedure Hold_display to demonstrate calls.            (See Example 5-6).   *}
{* ************************************************************************ *}

BEGIN { Main Program }

    { Get the size of the current window. }

    pad_$inq_windows ( stream_$stdout,
                       window_info,      { Current position of window }
                       max_windows,      { Maximum no. of windows desired }
                       n_windows,        { Number of windows open to pad }
                       status );
    check_status;

    { Get the width and height of current font. }

    pad_$inq_font ( stream_$stdout,
                    font_width,
                    font_height,
                    ' ',                 { No need to know name }
                    0,                   { No need to know name }
                    font_len,
                    status );
    check_status;
```

Example 5-13. Using PAD_$INQ_FONT

```
{ Set scale of window height and width to raster units. }

pad_$set_scale (stream_$stdout,
                1,                  { Scale factor of x-coordinate }
                1,                  { Scale factor of y-coorindate }
                status);
check_status;

pad_$create_frame ( stream_$stdout,
                    window_info[1].width, { Same size as window }
                    font_height,          { Same height as font height }
                    status );
check_status;

pad_$move ( stream_$stdout,
            pad_$absolute,
            5,                    { Raster units }
            font_height,          { Height of font }
            status );
check_status;

{ Put the prompt "#" in the input window with STREAM_$PUT_CHR. }

stream_$put_chr ( stream_$stdout,
                  ADDR( prompt_str ),    { Pointer to buffer }
                  SIZEOF( prompt_str ),  { Number of bytes to read }
                  seek_key,
                  status );
check_status;
hold_display;
 .
 .
 .
```

Example 5-13. Using PAD_$INQ_FONT (Cont.)

5.5.9. Sample Program: Creating a Window to Run a Clock

Examaple 5-14 uses miscellaneous PAD calls to create a digital clock. By default, it places the clock in the top left corner of the screen. The user can specify another position for the clock by specifying the x,y coordinates when the user executes the program.

This program also creates and uses a frame. For details on frames, see Section 5.7.

```
PROGRAM pad_digclk;

{ This program displays a digital clock on the screen. The user
  executes the program with the DM CPO command.  The user can optionally
  add the x,y coordinates on the command line to specify its location.
  Otherwise the clock runs in the top left corner of the screen. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';

CONST
  font_name    = 'f9x15.iv';     { Font file located in /sys/dm/fonts }
  window_num   = 1;
  as_time_len  = 8;
  border_size  = 5;
  close        = TRUE;
  no_border    = FALSE;

VAR
  status       : status_$t;
  window       : pad_$window_desc_t :=
                   [ 0, 0, 10, 10 ];  { Default window location }
  stream       : stream_$id_t;
  font_id      : integer;
  font_height  : integer;
  font_width   : integer;
  hunoz        : integer;
  hukairz      : integer;
  one_second   : time_$clock_t :=
                  [ high16 := 0,
                    low32 := 250000 ];
  now          :  cal_$timedate_rec_t;
  last_minute  : integer := -1;
  as_time      : ARRAY[1..as_time_len] OF char;  { ASCII time }
  key          : stream_$SK_t;

{* ********************************************************************** *}
{* Procedure Check_status for error handling.            (See Example 5-6).   *}
{* ********************************************************************** *}

{* ********************************************************************** *}
{* Procedure Get_num_arg checks to see if user provided arguments to specify  *}
{* the x,y coordinates of the clock. PGM_$GET_ARG returns a string, so convert *}
{* it to an integer. If all goes well, the result is assigned to arg_val.     *}
{* ********************************************************************** *}
```

Example 5-14. Using PAD Calls to Create a Clock

```
PROCEDURE get_num_arg ( arg_num: integer;
                        OUT arg_val: integer );

VAR
  arg       : string;
  argl      : integer;
  hunoz     : integer;
  hukairz   : integer;
  anyway    : integer;
  number    : integer;

BEGIN

    { Get argument from command line and assign its length to argl. }
    argl := pgm_$get_arg ( arg_num,        { Number of argument }
                           arg,            { Returns argument string }
                           status,         { Completion status }
                           SIZEOF(arg) ); { Max length of argument }

    IF status.all = status_$ok THEN
    BEGIN

        { Convert string to integer and assign to variable, hunoz }
        hunoz := vfmt_$decode2( '%wd%.',  { String }
                                arg,      { Text buffer }
                                argl,     { Size of text buffer }
                                hukairz,  { No need to know value }
                                status,   { Completion status }
                                number,   { Decoded data }
                                anyway ); { Decoded data }

        IF status.all = status_$ok THEN
            arg_val := number;
        END;
END; { get_num_arg }

BEGIN { Main Program }

    { Get window left coordinate, if user supplies it. }
    get_num_arg ( 1, window.left );

    { Get window top coordinate, if user supplies it.  }
    get_num_arg ( 2, window.top );

    { Create the window -- note that the size is 10x10 pixels, we
      will change it to after we know the font size. }
    pad_$create_window ( '',                 { Null pathname }
                         0,                  { Null namelength }
                         pad_$transcript,    { Type of pad }
                         1,                  { Number of display unit }
                         window,             { Position of window }
                         stream,             { Stream ID }
                         status );           { Completion status }
```

Example 5-14. Using PAD Calls to Create a Clock (Cont.)

```
check_status;

pad_$set_auto_close( stream, window_num, close, status );

{ Load the font and use it. }

pad_$load_font ( stream,
                 font_name,
                 SIZEOF(font_name),
                 font_id,                { Returns font ID }
                 status );
check_status;

pad_$use_font ( stream, font_id, status);
check_status;

{ Get the size of the font in use. }
pad_$inq_font ( stream,
                font_width,              { Returns width of font }
                font_height,             { Returns height of font }
                hunoz,                   { No need to know value }
                0,                       { No need to know value }
                hukairz,                 { No need to know value }
                status );
check_status;

{ Adjust window width and height to font size. }

window.width  := font_width * as_time_len + border_size;
window.height := font_height + border_size;

{ Make window borderless. }

pad_$set_border ( stream, window_num, no_border, status );
check_status;

{ Set scale to pixel values. }

pad_$set_scale ( stream, 1, 1, status );
check_status;

{ Set window to new size. }

pad_$set_full_window ( stream, window_num, window, status );
check_status;

{ Create a frame the same size as the window. }

pad_$create_frame ( stream, window.width, window.height, status );
check_status;
```

Example 5-14. Using PAD Calls to Create a Clock (Cont.)

```
WHILE TRUE DO
BEGIN    { Translate a system clock value into time value. }

    cal_$decode_local_time ( now );
    IF now.minute <> last_minute THEN
    BEGIN
        { If a minute has passed, clear the frame and write the
          minute and second value. Note that this happens the
          first time through. }

        pad_$clear_frame ( stream, 0, status );
        check_status;

        vfmt_$encode5 ( '%2wd:%2zwd:%2zwd%$', as_time, as_time_len,
                         hunoz, now.hour, now.minute, now.second, 0, 0 );

        { Put the output cursor at the left side of the frame. }
        pad_$move ( stream,
                    pad_$absolute,
                    border_size,
                    font_height,     { Must be at least font_height }
                    status );
        check_status;

        stream_$put_rec ( stream,
                          ADDR(as_time),
                          SIZEOF(as_time),
                          key,
                          status);
        check_status;

    END
    ELSE BEGIN  { Just write the seconds value. }

        vfmt_$encode2 ( '%2zwd%$', as_time, SIZEOF(as_time),
                        hunoz, now.second, 0 );

        { Move the output cursor to the 6th character position.
          Note that this only works with a fixed-sized font. }

        pad_$move ( stream,
                    pad_$absolute,
                    border_size+6*font_width,
                    font_height,
                    status );
        check_status;

        stream_$put_rec ( stream,
                          ADDR(as_time),
                          2,
                          key,
                          status);
        check_status;

    END;
```

Example 5-14. Using PAD Calls to Create a Clock (Cont.)

```
        last_minute := now.minute;

        time_$wait ( time_$relative, one_second, status );
        check_status;
    END;

END. { pad_digclk }
```

Example 5-14. Using PAD Calls to Create a Clock (Cont.)

5.6. Using Icons

The DOMAIN system allows users to represent a window in icon format so they can set a window aside without having to close its pad. You can use PAD calls to create a window in icon format, change a full-sized window to icon format, set the position of icons, and change the icon character displayed in the icon window.

Table 5-1 lists the PAD calls that create and manipulate icons.

Table 5-1. PAD System Calls to Create and Manipulate Icons

System Call	Operation
PAD_$CREATE_ICON	Creates a pad and window in icon format.
PAD_$MAKE_ICON	Changes an existing window into icon format.
PAD_$INQ_ICON	Returns information about a window in icon format.
PAD_$INQ_ICON_FONT	Returns information about the current icon font.
PAD_$ICON_WAIT	Waits until window is expanded from icon-format to full-window size, or until icon moves.
PAD_$SET_ICON_FONT	Sets the current icon font to a specified font name.
PAD_$SET_ICON_POSITION	Moves or sets an icon position for future use.

5.6.1. Creating an Icon

To get an icon, your program can either create a window in icon format, or change an existing window to icon format. To change a full-sized window into an icon, use the PAD_$MAKE_ICON system call. To create a window in icon format, use the PAD_$CREATE_ICON call.

Example 5-15 shows how to change a full-sized window into an icon using the PAD_$MAKE_ICON system call. The argument **stream_win** is the stream ID, in STREAM_$ID_T format, of the window you want to change to icon format. **Window_no** is the number of the specified window returned by PAD_$INQ_WINDOWS as described in Section 5.5.1.

Icon_char is the icon font character to be displayed in the window. You can either specify a character (such as "*") to get a specific icon character, or a blank character (' ') to use the default icon character for the type of pad. You can also specify a character from your own icon font by making a previous call to PAD_$SET_ICON_FONT, which is described in Section 5.6.3. **Status** is the completion status returned by the call.

```
pad_$make_icon ( stream_win,    { Stream of existing window }
                 window_no,     { Window number }
                 ' ',           { Default icon character }
                 status );      { Completion status }
```

Example 5-15. Changing a Window to an Icon

To create a new pad and window in icon format, use the PAD_$CREATE_ICON call. Example 5-16 shows how to create a new pad and window in icon format using this system call. You supply the **pathname** and **namelength**, and type of the pad you want to create. For details on these arguments, see Section 5.3.1.

Icon_pos is the location of the icon on the display, in PAD_$POSITION_T format. You set the values of the x and y coordinates before making this call, if you want to specify the icon's location on the display.

Icon_char is the icon font character to be displayed in the window. You can either specify a character (i.e., "*") to get a specific icon character, or a blank character (' ') to use the default icon character for the type of pad. You can also specify a character from your own icon font by making a call to PAD_$SET_ICON_FONT first. (See Section 5.6.3.)

Window indicates the size and position of the future window, in PAD_$WINDOW_DESC_T format. You set the values of window before making this call. **Stream_win** is the stream ID, in STREAM_$ID_T format, of the window you are creating. **Status** is the completion status returned by the call.

```
{ Set the position of the icon to the upper right corner. }
    icon_pos.x_cood := 1020;
    icon_pos.y_cood := 24;

{ Set location of future window. }
    window.top      := 500;
    window.left     := 500;
    window.width    := 500;
    window.height   := 500;
```

Example 5-16. Creating an Icon

```
{ Create a new window in icon format. }

    pad_$create_icon ( pathname,        { Pathname of pad }
                       namelength,      { Length of pathname }
                       pad_$edit,       { Type of pad }
                       display_unit,    { Number of display unit }
                       icon_pos,        { Location of icon on display }
                       icon_char,       { Icon font character displayed }
                       window,          { Location of future window }
                       stream_edit,     { Stream ID }
                       status );        { Status code }
```

Example 5-16. Creating an Icon (Cont.)

Once you have created an icon, you can change the icon to its associated window with the PAD_$SELECT_WINDOW call.

5.6.2. Positioning an Icon

Oftentimes, a user has a particular place on the display for icons. Your program can check to see if the user moved the icon, then places the icon in this same position the next time it is created. The PAD_$ICON_WAIT system call automatically checks to see if the icon has been moved.

You can also change the position of the icon, or replace the current icon character, by using the PAD_$SET_ICON_POSITION call. If the window specified is already in icon format, then the call moves the icon to the new location. If you want to change either the position or the icon character without changing the other, use PAD_$INQ_ICON first to determine the information that is not changing.

Example 5-17 is a program that uses these system calls to place the icon where the user wants it. It also uses PAD_$SET_ICON_POSITION to change the icon character in use.

```
PROGRAM pad_make_icon;

{ This program is a sample of using icons. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit = 1;
  auto_close   = TRUE;
```

Example 5-17. Changing Icon Position and Character

```
VAR
  stream_win   : stream_$id_t;
  pane_stream  : stream_$id_t;
  seek_key     : stream_$sk_t;
  status       : status_$t;

  window       : pad_$window_desc_t;  { Position, height, width of window }
  window_no    : integer;             { Number of windows open to a pad }
  window_list  : pad_$window_list_t;  { Array of up to 10 windows }
  window_size  : integer;             { Maximum no. of windows desired }

  icon_pos     : pad_$position_t;     { Position of icon }
  icon_char    : char;
  icon_moved   : boolean := FALSE;    { Checks if icon moved }

{* ********************************************************************** *}
{* Procedure Check_status for error handling.          (See Example 5-6).  *}
{* ********************************************************************** *}
{* Procedure Hold_display to demonstrate calls.        (See Example 5-6).  *}
{* ********************************************************************** *}

BEGIN { Main Program }
    .
    . { Set postion of future windows. }
    .

    { Set position of icon to upper right corner.  }
    icon_pos.x_coord := 1020;
    icon_pos.y_coord := 24;

    . { Create a new transcript window using pad_$create_window. }
    . { Get window statistics for next calls with pad_$inq_windows. }
    . { Make window close when stream closes with pad_$set_auto_close. }
    . { Do work in window ... }

    { Change window into an icon. }

    icon_char := ' ';
    pad_$make_icon ( stream_win,  { Stream ID }
                     window_no,   { Window number }
                     icon_char,   { Default character icon }
                     status );    { Completion status }
    check_status;

    { Move position of icon and change the icon character. }

    icon_pos.x_coord := 950;
    icon_pos.y_coord := 710;
    icon_char        := '*';

    pad_$set_icon_pos ( stream_win, { Stream ID }
                        window_no,  { Window number }
                        icon_pos,   { Position of icon }
                        icon_char,  { Icon character }
                        status );   { Completion status }
    check_status;
```

Example 5-17. Changing Icon Position and Character (Cont.)

```
    { Suspend process until user expands window from icon format. }

    pad_$icon_wait ( stream_win,
                     window_no,
                     icon_moved,  { TRUE if icon moved }
                     icon_pos,    { Returns new position of icon }
                     status );
    check_status;
    hold_display;

    { Turn transcript window into an icon. }

    pad_$make_icon ( stream_win,
                     window_no,
                     icon_char,
                     status );
    check_status;
    hold_display;

    { Close stream with stream_$close. }

END.    { pad_make_icon }
```

Example 5-17. Changing Icon Position and Character (Cont.)

5.6.3. Creating Your Own Icon Font

You can determine which icon character will be displayed in the icon window by using the active icon character set or supplying your own character set.

If you use the active icon character set, you can either specify which character you want, or specify the blank character to get the default character for the type of window pad specified.

The default icon character set is contained in the font file /SYS/DM/FONTS/ICONS. You can edit this font file to create your own icon characters by using the font editor, EDFONT.

You can also use EDFONT to create your own icon font file, and then use the PAD_$SET_ICON_FONT call to supply its pathname. You can use PAD_$INQ_ICON_FONT to get the pathname of the current icon font before replacing it with your own font's pathname, so that you can restore the original font before terminating your program. For a complete description of EDFONT, see the *DOMAIN System Command Reference* manual.

5.6.4. Sample Program: Using Icons

Example 5-18 is a sample program using various PAD calls to create and manipulate icons. It creates a new window with an input pad in icon format. It uses STREAM_$PUT_CHR to put a prompt in the input pad, and a STREAM_$GET_REC to get input from the keyboard.

```
PROGRAM pad_create_icon;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit = 1;
  prompt_str   = ('# ');
  auto_close   = TRUE;

VAR
  stream_win    : stream_$id_t;
  pane_stream   : stream_$id_t;
  seek_key      : stream_$sk_t;
  status        : status_$t;

  window        : pad_$window_desc_t;  { Position, height, width of window }
  window_no     : integer;             { Number of windows open to a pad }
  window_list   : pad_$window_list_t;  { Array of up to 10 windows }
  window_size   : integer;             { Maximum no. of windows desired }

  icon_pos      : pad_$position_t;     { Position of icon }
  icon_char     : char;
  icon_moved    : boolean;             { Indicates whether user moved icon }

  buffer        : string;              { Buffer to hold keyboard input }
  return_ptr    : ^string;
  return_len    : integer32;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.           (See Example 5-6).   *}
{* ************************************************************************ *}
{* Procedure Hold_display to demonstrate calls.         (See Example 5-6).   *}
{* ************************************************************************ *}

BEGIN { Main Program }

    { Set position of future windows. }

    window.top    := 10;
    window.left   := 10;
    window.width  := 300;
    window.height := 300;

    { Set position of icon to upper right corner.  }

    icon_pos.x_coord := 1020;
    icon_pos.y_coord := 24;
```

Example 5-18. Using Icons

```
{ Create a new transcript window in icon format.  The icon will have
  the Shell icon character from the default icon font. }

icon_char := '*';

pad_$create_icon ( ' ',                  { No pathname for transcript pad }
                   0,                    { No namelength }
                   pad_$transcript,      { Type of pad }
                   display_unit,         { Which display unit -- 1 }
                   icon_pos,             { Location -- x and y coordinates }
                   icon_char,            { Icon font displayed }
                   window,               { Location of future window }
                   stream_win,           { Stream ID of new window }
                   status );             { Completion status. }
check_status;

{ Create an input pad for the new transcript pad. This is a
  window pane associated with the same window. }

pad_$create ( ' ',                       { No pathname for input pad }
              0,                         { No namelength for input pad }
              pad_$input,                { Type of pad }
              stream_win,                { Stream ID of related transcript pad }
              pad_$bottom,               { Input pads always go on bottom }
              [],                        { Pane_size is relative }
              20,                        { New pad takes up 20% of related window }
              pane_stream,               { Stream ID of this input pad }
              status );                  { Completion status }
check_status;

{ Get window statistics for next calls. }

pad_$inq_windows ( stream_win,           { Stream ID }
                   window_list,          { Location, size of window }
                   window_size,          { Max number of windows desired }
                   window_no,            { Number of windows open to pad }
                   status );             { Status code }
check_status;

{ Make window close when stream closes. }

pad_$set_auto_close ( stream_win,
                      window_no,
                      auto_close,
                      status );
check_status;

{ Suspend process until user opens icon. It checks to see if icon
  has moved.  If it has, it moves the icon to the new position
  when it returns to an icon. }

writeln ( 'Process suspended until user turns icon into window, ' );
writeln ( 'or until user moves the icon. If user turns icon into ' );
writeln ( 'a window, it waits for input.  After user types input,' );
writeln ( 'it waits 3 seconds, then turns the window into an icon. ' );
```

Example 5-18. Using Icons (Cont.)

```
    pad_$icon_wait ( stream_win,
                     window_no,
                     icon_moved,        { TRUE if icon moved. }
                     icon_pos,          { If TRUE, new position of icon. }
                     status );
    check_status;

    { Put the prompt "#", in the input window with STREAM_$PUT_CHR. }

    stream_$put_chr ( stream_win,               { Stream of transcript pad }
                      ADDR( prompt_str ),       { Pointer to buffer }
                      SIZEOF( prompt_str ),     { Number of bytes to read }
                      seek_key,
                      status );
    check_status;

    { Get information from input pad with STREAM_$GET_REC. }

    stream_$get_rec ( pane_stream,
                      ADDR( buffer ),        { Buffer holding input}
                      SIZEOF( buffer ),
                      return_ptr,            { Return pointer }
                      return_len,            { Return length }
                      seek_key,              { Seek key }
                      status );              { Completion status }
    check_status;
    hold_display;

    { Turn transcript window into an icon. }

    pad_$make_icon ( stream_win,
                     window_no,
                     icon_char,
                     status );
    check_status;
    hold_display;

    { Now, program turns window from icon format to full-sized window. }

    writeln ( 'The program will now automatically turn the window ' );
    writeln ( 'from icon format to full-sized window, and then terminate. ' );

    pad_$select_window ( stream_win,
                         window_no,
                         status );
    check_status;
    hold_display;

    stream_$close ( stream_win,
                    status );
    check_status;

END.    { pad_create_icon }
```

Example 5-18. Using Icons (Cont.)

5.7. Handling Graphics Input with Frames

Usually, your program output can be displayed on single lines of text. In this case, the program can reposition the output cursor only horizontally from the beginning to the end of the line. In some cases though, you may need to display more information than can fit on a line, and you may want to move the cursor up and down as well as right to left. You can have this control when you create a frame with the PAD_$CREATE_FRAME system call.

A **frame** is an area within a transcript pad where the cursor can move anywhere. As Example 5-1 illustrates, the debugger uses two frames in its display: one holds the pathname of the target program, the other holds the source line numbers and an arrow pointing to the current line.

The most common reason for creating frames is for handling two-dimensional text output, in the style of a *dumb terminal*. You can also get two-dimensional input in a frame. For more complex graphics using I/O, see the *Programming with DOMAIN Graphics Primitives* manual.

NOTE: If you use GPR in frame mode for graphics input, GPR uses PAD calls to create and manipulate frames. Therefore, you cannot use the following PAD calls in the same program: PAD_$CREATE_FRAME, PAD_$CLEAR_FRAME, PAD_$CLOSE_FRAME, PAD_$DELETE_FRAME, PAD_$SET_SCALE, PAD_$LOAD_FONT or PAD_$USE_FONT.

If you use GPR in direct mode, there are even more restrictions on using PAD system calls. For details, see the *Programming with DOMAIN Graphics Primitives* manual.

5.7.1. Creating the Frame

You can create a frame in any transcript pad. If you create a frame on a new transcript pad, it fills the entire transcript window. (The user can still scroll back to see the previous contents of the transcript pad.) If your application is mostly graphics, you are more likely to use the original pad.

Example 5-19 shows how to create a frame. You specify the stream ID of an existing transcript pad, and the width and height of the new frame, scaled according to current scale factors.

Note that the Display Manager clips output to the frame size you specify. You will get the error "value out of range" if you try to position the cursor outside the frame. Since there are no efficiency penalties related to the size of the frame, you can simply create the maximum size frame available (32767 x 32767 raster units), if you want.

```
CONST
  max_frame_sz = 32767;

  pad_$create_frame ( stream_trans, { Stream ID of existing transcript pad }
                      max_frame_sz, { Width of new frame in pixels (scaled) }
                      max_frame_sz, { Height of new frame in pixels (scaled) }
                      status );     { Completion status }
```

Example 5-19. Creating a Frame

If you create a frame of the maximum size, you can use the calls PAD_$INQ_VIEW and PAD_$SET_VIEW to position the pad over the part of the frame you want.

For example, your program may create a frame larger than the entire display to contain a large picture, such as a mechanical drawing. You can allow the user to view pieces of the picture at a time. If you want the user to have easy access to a particular part of the picture, such as the title block, you can use PAD_$SET_VIEW to move the window over the title block.

The DOMAIN Language Level Debugger is another example of using PAD_$INQ_VIEW and PAD_$SET_VIEW. If the user directs the debugger to a specified line number, the debugger checks to see if the line number is already in view with PAD_$INQ_VIEW. If not, it uses PAD_$SET_VIEW to move the window over the desired line number.

You can control output in a frame by using PAD calls that manipulate the output cursor. For details, see Section 5.8.3.

5.7.2. Clearing the Frame

When you move the cursor to a certain position in the frame and write text there, you can make the old text seem to disappear. But it doesn't actually go away; it is still *underneath* the new text. For example, when a user debugs a program, the arrow moves up and down the frame to point to the current line. Each time the arrow moves, the Display Manager merely overwrites the frame with the arrow's new location; its previous locations still exist underneath.

If your program frequently overwrites text in a frame, you should use the PAD_$CLEAR_FRAME call. This deletes all the text ever written, up to a point specified in the call. If you specify zero, it deletes all the text. If you do not clear the frame yourself, and the window needs to be redrawn (for example, due to popping windows), the redraw procedure will be quite lengthy.

When you are finished with the frame, you call PAD_$CLOSE_FRAME. This closes the frame, leaving the final image of the frame in the transcript pad, and returns the pad to line mode. If you do not want this image on your transcript pad, you can call PAD_$DELETE_FRAME instead. Note that this deletes the frame from the transcript pad altogether, so you have no record of the text within the frame.

5.7.3. Sample Program: Creating and Writing to Frames

Example 5-20 is a program example that uses PAD calls to create and clear a frame. It also uses other PAD calls described previously in this chapter.

This program creates a frame at the top of a window, and displays the name of a file in the inverted version of the current font. It uses PAD_$INQ_FONT to get the name of the font, and adds the .iv extension with the stringcopy function. Then it creates an edit pad under the frame.

```
PROGRAM pad_filename;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';

CONST
  display_unit       = 1;
  window_count       = 1;
  auto_close         = TRUE;
  pane_size          = 1;
  max_frame_size     = 32767;

TYPE
  bufstring          = ARRAY [1..512] OF CHAR;   { String buffer }

VAR
  source_stream      : stream_$id_t;
  pane_stream        : stream_$id_t;
  pane_edit_stream   : stream_$id_t;
  seek_key           : stream_$sk_t;

  window             : pad_$window_desc_t;
  window_list        : pad_$window_list_t;
  window_size        : integer;
  frame_width        : integer;
  frame_height       : integer;

  status             : status_$t;
  pathname           : name_$pname_t;
  namelength         : integer;
  count              : integer;
  source_name_font   : static integer := -1;
  inverted_font_name : pad_$string_t;    { Buffer to make inverted name. }
  font_heigth        : integer;
  font_width         : integer;
  font_len           : integer;          { Size of font returned }
  font_name          : PAD_$STRING_T;
  i                  : integer;

{* ************************************************************************ *}
{* Function stringcopy copies a given string to a buffer, and returns the   *}
{* number of characters to be copied.  It stops at the character pair, %$.  *}
{* ************************************************************************ *}

FUNCTION stringcopy ( IN src  : UNIV bufstring;
                      OUT dst : UNIV bufstring) : integer;

VAR
    i, j : integer;          { Indexes to src and dst strings }
```

Example 5-20. Displaying a Filename at the Top of a File

```
BEGIN { stringcopy }

    i := 1;                  { Initialize the indexes  }
    j := 1;

    WHILE (src[i] <> '%') OR (src[i+1] <> '$') DO
        BEGIN
            dst[j] := src[i];
            i := i + 1;
            j := j + 1;
        END;

    stringcopy := j - 1;      { The number of characters copied. }

    RETURN;

END; { stringcopy }

{* ************************************************************************* *}
{* Procedure Check_status for error handling.            (See Example 5-6).   *}
{* ************************************************************************* *}
{* Procedure Hold_display to demonstrate calls.          (See Example 5-6).   *}
{* ************************************************************************* *}

BEGIN { Main Program }

    { Set position of future window.  }

    window.top    := 10;
    window.left   := 10;
    window.width  := 500;
    window.height := 500;

    { Get pathname from keyboard and set values of pathname, namelength. }

    writeln ('Type in the pathname of the file: ');

    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);
    check_status;

    { Create original transcript pad and window. }

    pad_$create_window( '',                { No pathname for transcript pad }
                        0,                 { No namelength for transcript pad }
                        pad_$transcript,   { Type of pad }
                        display_unit,      { Number of display unit }
                        window,            { Position of window }
                        source_stream,     { Returns stream ID }
                        status);           { Completion status }
    check_status;
```

Example 5-20. Displaying a Filename at the Top of a File (Cont.)

```
{ Close window when stream closes. }

pad_$set_auto_close ( source_stream,  { Stream ID }
                      window_count,   { Number of window }
                      auto_close,     { Flag -- set to TRUE }
                      status);        { Completion status }
check_status;

{ Make a transcript pad and window pane for the name of file. }

pad_$create ( '',                { No pathname  }
              0,                 { No namelength }
              pad_$transcript,   { Type of pad }
              source_stream,     { Same stream ID as window above }
              pad_$top,          { Location of new window pane }
              [pad_$abs_size],   { Pane size is absolute value }
              pane_size,         { Pane height is 1 line }
              pane_stream,       { Stream ID of window pane }
              status );          { Completion status }
check_status;

{ Close window when stream closes. }

pad_$set_auto_close ( pane_stream,
                      window_count,
                      auto_close,
                      status);
check_status;

{ Now make frame in above pad to hold inverted pathname. }

frame_width  := max_frame_size;
frame_height := pane_size;

pad_$create_frame ( pane_stream,    { Same as window pane }
                    frame_width,    { Same as window pane }
                    frame_height,   { Same as window pane }
                    status );
check_status;
hold_display;

{ Before printing the filename, find out the inverted font name of
  the font name in use. }

inverted_font_name := (' ');

pad_$inq_font ( source_stream,     { Stream ID of original transcript pad }
                font_width,        { Returns width of font }
                font_height,       { Returns height of font }
                font_name,         { Returns name of font }
                SIZEOF(font_name),{ Size of buffer for font_name }
                font_len,          { Length of font_name }
                status );          { Completion status }

inverted_font_name := font_name;  { Copy to working buffer }
```

Example 5-20. Displaying a Filename at the Top of a File (Cont.)

```
{ Assume font is not bold, try loading the bold inverted
  version of the same font by adding the extension (".b.iv")
  to the font_name with the stringcopy function. }

i := font_len +
     stringcopy('.b.iv%$', inverted_font_name[font_len + 1]);

pad_$load_font ( pane_stream,        { Stream of frame }
                 inverted_font_name, { Font_name + ".b.iv" }
                 i,                  { Length of font_name }
                 source_name_font,   { Returns font ID }
                 status);            { Completion status }

{ If the font is already bold, it returns an error, so try
  adding the inverted extension (".iv") only. }

IF status.all <> 0 THEN
    BEGIN
        i := font_len +
             stringcopy ('.iv%$', inverted_font_name[font_len + 1]);

        pad_$load_font ( pane_stream,
                         inverted_font_name,
                         i,
                         source_name_font,
                         status);

        IF status.all <> 0 THEN
        BEGIN
            source_name_font := 0;    { Use the default font. }
            status.all := status_$ok;
        END;
    END;

{ Now clear the frame to erase any old filenames, and
  write the new name. }

pad_$clear_frame ( pane_stream,
                   0,                  { Clear entire frame }
                   status );
check_status;

{ Use PAD_USE_FONT to have program use the desired font. }

IF source_name_font <> 0 THEN

BEGIN
    pad_$use_font ( pane_stream,       { Stream of frame }
                    source_name_font,  { Font ID returned above }
                    status );
    status.all := status_$ok;
END;
```

Example 5-20. Displaying a Filename at the Top of a File (Cont.)

```
    { Put output cursor in frame. }

    pad_$move ( pane_stream,
                pad_$absolute,  { Move relative to top left of frame }
                5,              { x coordinate relative to frame }
                1,              { y coordinate relative to frame }
                status );
    check_status;

    { Write name of file in frame. }

    stream_$put_rec ( pane_stream,
                      ADDR(pathname),
                      namelength,
                      seek_key,
                      status );
    check_status;
    hold_display;

    { Make an edit pane for the rest of the window below the frame, and
      associate it with specified file. }

    pad_$create ( pathname,
                  namelength,
                  pad_$edit,
                  source_stream, { Same stream ID as window }
                  pad_$bottom,   { New pane position on original pad }
                  [],            { Pane height relative to original pad }
                  100,           { Height = 100% of original pad, minus frame. }
                  pane_edit_stream,   { Returns stream ID of window pane }
                  status );
    check_status;

    { Close edit pad when stream closes. }

    pad_$set_auto_close ( pane_edit_stream,
                          window_count,
                          auto_close,
                          status);
    check_status;

    { Close the streams. }

    stream_$close( pane_edit_stream, status );
    check_status;

    stream_$close( pane_stream, status ;
    check_status;

    stream_$close( source_stream, status );
    check_status;

END.   { pad_filename }
```

Example 5-20. Displaying a Filename at the Top of a File (Cont.)

5.8. Sending and Receiving Program Input

To handle input and output, most programs use an input pad and the STREAM system calls described in Chapter 4. In this case, the operating system reads text from the keyboard, buffers it in the input pad (so the user can edit the line), and then copies it to the transcript pad (when the user hits <RETURN>). Your program reads from the input pad.

Sometimes, you might want to bypass any system processing, for example, to prevent the system from echoing any input on the display. Your program can read the keyboard input directly if you put the input pad in *raw mode*. Section 5.8.1 describes getting and receiving input in the normal, *cooked mode*. Section 5.8.2 describes how to bypass system input in raw mode.

5.8.1. Processing System Input in Cooked Mode

Normally, when your program receives keyboard input, it buffers it in the input pad to allow the user to edit it before submitting it to the program by pressing <RETURN>. This is called **cooked mode** processing because the display manager *cooks* (or preprocesses) the keyboard input by displaying each keystroke in the input pad. Cooked mode allows the user to edit the input before signaling the program to read it by pressing <RETURN>. It then copies the text from the input pad to the transcript pad.

Every input pad starts out in cooked mode unless you create the input pad with the [PAD_$INIT_RAW] option to initialize it in raw mode.

When you exchange data with the Display Manager, you usually do it in terms of stream records. A **stream record** is usually a string of visible text with the NEWLINE character marking its end. Stream records can contain any character, including control characters (such as NEWLINE or form feed) at any character position. The Display Manager limits stream records to 256 characters in length.

Some stream calls deal with incomplete records or single characters. When your program sends partial data to the Display Manager through stream calls to standard output, the Display Manager buffers the partial text. It becomes visible in the transcript pad only when you issue a stream call to complete the record.

For example, if you write an incomplete record to the transcript pad and then ask for an input record, the Display Manager moves the incomplete record to the input pad as a prompt that tells the user what to type. When the user types a record and presses RETURN, the Display Manager moves the complete record (your prompt and the user's response) to the transcript pad. The user's response becomes the input record for your program.

5.8.2. Bypassing System Input Processing with Raw Mode

In **raw mode**, the Display Manager does not buffer keystrokes in the input pad, nor does it echo them in the transcript pad. Actually, the input window goes away, and the keyboard cursor is tied to the transcript pad's output cursor when the cursor is in the transcript window. A common use for raw mode is to ask for a user's password without recording it in the transcript pad.

The program can also read the keyboard cursor position at each keystroke if you use the PAD_$CPR_ENABLE call. This is most useful for graphics input. (However, in most cases, you will want to use GPR rather than PAD calls for graphics input.)

In raw mode, you can call STREAM_$GET_REC or STREAM_$GET_BUF to get the characters that the user typed at the time of the call. It gets as many characters as the limit you specified in the call.

Example 5-21 is a program that uses the PAD_$RAW call to request the user's password without having it echo in the transcript pad. When you are done using raw mode, be sure to return the pad to normal, or cooked mode.

```
PROGRAM pad_raw_mode;


{ This program shows how to use raw mode. It asks for your
  password but does not echo the input to the screen.  After
  you type in your password, it replies,"Thank you." }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';


CONST
  display_unit = 1;
  auto_close   = TRUE;
  message      = ( ' Enter your password: ');
  reply        = ( ' Thank you. ');
  window_no    = 1;


VAR
  stream_one   : stream_$id_t;
  pane_stream  : stream_$id_t;
  seek_key     : stream_$sk_t;
  status        : status_$t;

  window_one    : pad_$window_desc_t;
  window_list   : pad_$window_list_t;

  move_char    : integer;
  buffer       : string;        { Buffer to hold keyboard input }
  return_ptr   : ^string;
  return_len   : integer32;
  i            : integer;


{* ********************************************************************** *}
{* Procedure Check_status for error handling.             (See Example 5-6).   *}
{* ********************************************************************** *}
```

Example 5-21. Using Raw Mode

```
BEGIN { Main Program }
    .
    .
    .
    { Create an input pad and initialize it in raw mode. }

    pad_$create ( ' ',
                  0,
                  pad_$input,
                  stream_one,
                  pad_$bottom,
                  [pad_$init_raw],
                  20,
                  pane_stream,
                  status);
    check_status;

    { Write message to the transcript pad. }

    stream_$put_rec ( stream_one,
                      ADDR( message ),
                      SIZEOF( message ),
                      seek_key,
                      status);
    check_status;

    { Get input from keyboard.  It gets each character until it
      reaches a carriage return.  }

    i := 1;

    REPEAT

        stream_$get_rec ( pane_stream,         { Standard input -- keyboard }
                          ADDR( buffer[i] ),   { Buffer holding input}
                          SIZEOF( buffer ) -i + 1,
                          return_ptr,          { Return pointer }
                          return_len,          { Return length }
                          seek_key,            { Seek key }
                          status );            { Completion status }

        check_status;
        i := i + return_len;

    UNTIL buffer[i - 1] = CHR(pad_$cr);

    { Move output cursor to where the message text ends. }

    move_char := sizeof (message) + 1;

    pad_$move   ( stream_one,
                  pad_$absolute,
                  move_char,
                  1,
                  status);
    check_status;
```

Example 5-21. Using Raw Mode (Cont.)

```
    { Write reply in window. }

    stream_$put_rec ( stream_one,
                      ADDR( reply ),
                      SIZEOF( reply ),
                      seek_key,
                      status);
    check_status;

    { Return to normal cooked processing before closing stream. }

    pad_$cooked ( pane_stream, status );
    check_status;

    stream_$close ( stream_one, status);
    check_status;

END. { pad_raw_mode }
```

Example 5-21. Using Raw Mode (Cont.)

5.8.3. Controlling System Output with Cursors

To control output in a frame, you can use PAD calls that manipulate the output cursor. Each transcript pad has an invisible **output cursor** that points to the position where the next program output will appear. You control the position of the output cursor with the PAD_$MOVE system call.

You can also have indirect control over the keyboard cursor if your input pad is in raw mode. Each display has a visible **keyboard cursor** that indicates where the next typed character will appear. The keyboard cursor is a blinking rectangle, or in touchpad mode, a small arrow. The user controls the position of the keyboard cursor with Display Manager commands. If the user moves the keyboard cursor to the corresponding transcript pad, the keyboard cursor follows the output cursor each time the program sends output to the transcript pad.

In raw mode, your program can use the PAD_$LOCATE and PAD_$CPR_ENABLE calls to get the location of the keyboard cursor each time the user types a character. Example 5-22 shows how to use PAD_$CPR_ENABLE to report cursor positions in raw mode.

```
PROGRAM pad_cpr_enable;

{ This program turns the user's standard input into raw mode, waits for
  user to type a character, then reports the character position. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/vfmt.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
```

Example 5-22. Using PAD_$CPR_ENABLE to Report Cursor Positions

```
CONST
  display_unit   = 1;
  max_frame_size = 32767;

TYPE
  { Use this record to receive input.  Normally, you would get
    cursor position reports by finding the flag in a stream of
    data from the keyboard.  This record allows for efficient
    handling of a single CPR. }

  report = PACKED RECORD
     flag      : 0..255;      { Should be 16#FF }
     xhi, xlo  : 0..255;      { Integer }
     yhi, ylo  : 0..255;      { Integer }
     text      : char;
  END; { type }

VAR
  stream_in     : stream_$id_t := stream_$stdin;
  stream_out    : stream_$id_t := stream_$stdout;
  seek_key      : stream_$sk_t;
  status        : status_$t;

  return_len    : integer32;
  bufptr        : ^report;
  return_ptr    : ^report;
  report_buf    :  report;
  ix            :  integer;
  iy            :  integer;
  outbuf        :  array [1..2] of char;

{* ************************************************************************ *}
{* Procedure Check_status for error handling.          (See Example 5-6).   *}
{* ************************************************************************ *}

BEGIN { Main Program }

    { Create a frame on user's transcript pad to read cursor position
       reports. }

    pad_$create_frame  ( stream_out,
                         max_frame_size,
                         max_frame_size,
                         status );
    check_status;

    { Change input pad to raw mode to get cursor position reports. }

    pad_$raw ( stream_in, status );
    check_status;

    {  Get a cursor position report for each keystroke. }

    pad_$cpr_enable ( stream_in, PAD_$CPR_ALL, status );
    check_status;
```

Example 5-22. Using PAD_$CPR_ENABLE to Report Cursor Positions (Cont.)

```
    { Get input from keyboard. }

    stream_$get_rec ( stream_in,            { Standard input -- keyboard }
                      ADDR(report_buf),     { Buffer holding input}
                      SIZEOF(report_buf),   { Size of buffer }
                      return_ptr,           { Return pointer }
                      return_len,           { Return length }
                      seek_key,             { Seek key }
                      status );             { Completion status }
    check_status;

    { X and Y must be integers aligned on word boundaries.  Since they follow
      a Boolean in the record, they are not aligned. X and y are defined as an
      array of 255 integers so they can be aligned. }

    WITH return_ptr^ DO
        BEGIN
            ix := xhi  * 256 +  xlo;
            iy := yhi  * 256 +  ylo;
        END;

    { Move the output cursor to where the user put the input cursor. }

    pad_$move  ( stream_out,
                 pad_$absolute,
                 ix,
                 iy,
                 status);
    check_status;

    { This is the character the user entered.  The NEWLINE character marks
      the end of the display record. }

    outbuf[1]  := return_ptr^.text;
    outbuf[2]  := CHR(pad_$newline);

    { Write to the keyboard the character that the user typed. }

    stream_$put_rec ( stream_out,
                      ADDR ( outbuf ),
                      SIZEOF ( outbuf ),
                      seek_key,
                      status);
    check_status;

    { Close frame and return to cooked mode before program exits. }

    pad_$close_frame ( stream_out, status );
    check_status;

    pad_$cooked ( stream_in, status );
    check_status;

END.    { pad_cpr_enable }
```

Example 5-22. Using PAD_$CPR_ENABLE to Report Cursor Positions (Cont.)

You can also control the cursor's position by redefining the arrow keys (with PAD_$DEF_PFK) on the user's keyboard, so that they can signal your program rather than invoke Display Manager commands. If your program is in raw mode, your program can respond to these keystrokes by moving the cursors. The user can still move the cursor by using the mouse or touchpad.

5.8.4. Writing to an Output Stream: Control Codes and Escape Sequences

When your program writes to an output stream under the control of the Display Manager, ASCII characters (codes from 32 to 126 decimal) instruct the Display Manager to produce the visible character that corresponds to the code. The Display Manager refers to the current character font to determine the appearance of the visible character.

Some ASCII characters (codes from 0 to 31 decimal) do not correspond to a particular character, but rather to a control code. A **control code** tells the Display Manager to take a formatting action on the window or window pane in which they are sent. Table 5-2 lists these special actions.

Table 5-2. Control Codes to Format Output to Windows and Panes

Name	ASCII Character (Decimal)	Description
PAD_$CR	13	Moves the cursor to the start of the same line it is on.
PAD_$ESCAPE	27	Introduces a literal: the Display Manager does not interpret the next character.
PAD_$FF	12	Makes subsequent output start at the top of the window or window pane.
PAD_$NEWLINE	10	Marks the end of an input or output line; makes subsequent text start on a new line.
PAD_$TAB	9	Moves the cursor to the next tab stop.
PAD_$BS	8	Moves the cursor one character position to the left, if there is room in the window. (This is meaningful only if the current font has characters of the same width.)

To prevent the Display Manager from interpreting a control code literally, it can be preceded with the PAD_$ESCAPE character. Instead of performing the control code, the Display Manager writes the control code literally (if the current font has a character corresponding to that control code).

In certain cases, the PAD_$ESCAPE character introduces a multicharacter sequence. The Display Manager supports certain escape sequences, according to the ANSI standard. When you write such an escape character to a stream controlled by the Display Manager, it takes a special effect on the line or frame where the output cursor is located. These are useful alternatives to some Display Manager calls.

When you use escape sequences in lines instead of frames, the Display Manager ignores the line parameter, and the action occurs on the current line (pointed to by the output cursor).

Table 5-3 lists the multicharacter escape sequences. The *ESC* stands for the character PAD_$ESCAPE.

Table 5-3. Escape Sequences

Control Sequence	Description
ESC[line;columnH	Moves the cursor to the specified line and column. If used outside a frame, the Display Manager ignores the line parameter, and moves the cursor to the specified column of the current line.
ESC[OK	Erases characters in the current line, from the output cursor to the end of the line.
ESC[1K	Erases characters in the current line, from the start of the line to the output cursor.
ESC[2K	Erases the entire curent line.
ESC[OJ	Erases character positions in the frame, from the output cursor to the end of the line.
ESC[1J	Erases character positions in the frame, from the start of the line to the output cursor.
ESC[2J	Erases the entire frame.

5.9. Using Paste Buffers

Paste buffers are stream files located in the directory 'NODE_DATA/PASTE_BUFFERS. During your program's execution, you can use paste buffers to hold text or graphic images that a user *cuts* from one part of the pad, and intends to *paste* into the same or different pad. You can think of them as *clipboards* for your users to hold information temporarily.

5.9.1. Reading and Writing to Paste Buffers

Users gain access to paste buffers by using the Display Manager commands for copy (XC), cut (XD), and paste (XP). Programs gain access to paste buffers by using the system calls PBUFS_$CREATE and PBUFS_$OPEN, then reading or writing the contents of the file using stream calls.

Programs can also use the PAD_$DM_CMD to invoke a keyboard-style Display Manager command that cuts or pastes text, specifying a particular paste buffer.

When you create a paste buffer, you must refer to it by name. The name of the paste buffer is the object name of the buffer file in the directory 'NODE_DATA/PASTE_BUFFERS. Since the paste buffers always reside in this directory, paste buffer calls do not allow you to specify a full pathname as the name of the paste buffer. The name must be 32 characters, padded with blanks.

Each paste buffer can hold either text or image data. Text paste buffers are simply UASC stream files, and can be read with stream calls. Image paste buffers are essentially graphics map files (GMF). For details on GMF files, see the *Programming with DOMAIN Graphics Primitives* manual. When you create the paste buffer, you must specify whether it contains type or graphic images. Once created, you must use the buffer according to its type.

All paste buffers (that is, all files in 'NODE_DATA/PASTE_BUFFERS) are temporary, and go away when your program terminates, or when the user logs out.

5.9.2. Sample Program: Using Paste Buffers

Example 5-23 is an example of a program using PBUFS calls. It asks the user to supply the name of the paste buffer. If it exists, it writes the contents of the buffer. If it does not exist, it reads lines of input from the keyboard until the user types CTRL/Z. It repeats the sequence, asking the user to supply names of paste buffers until the user types STOP.

```
PROGRAM pbufs_paste_buffer(input, output);

{ This program manipulates paste buffers. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/name.ins.pas';
%INCLUDE '/sys/ins/pbufs.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';

CONST
  text        = TRUE;
VAR
  stream_buf  : stream_$id_t;
  status      : status_$t;
  info        : name_$pname_t;
  buffer_name : name_$pname_t;
  seek_key    : stream_$sk_t;
  buflen      : integer32;
  retptr      :^name_$pname_t;
  retlen      : integer32;
  done        : boolean;

{* ********************************************************************** *}
{* Procedure Check_status for error handling.          (See Example 5-6).  *}
{* ********************************************************************** *}
```

Example 5-23. Using Paste Buffers

```
PROCEDURE error_routine;        { for error handling }

BEGIN
    pgm_$set_severity( pgm_$error );
    pgm_$exit;
END;  { error_routine }

{* ************************************************************************ *}

BEGIN  { MAIN PROGRAM }

    { Write initial prompt }

    done := FALSE;

    writeln ( '          =================                ');
    writeln ( ' Type the name of the paste buffer: ');
    writeln ( ' Or type STOP to quit. ');
    writeln;

    readln  ( buffer_name );
    IF ( buffer_name = 'STOP') OR ( buffer_name = 'stop')
       THEN done := TRUE;

    WHILE NOT done DO

    BEGIN

        { Open existing paste buffer and write contents to screen. }

        pbufs_$open ( buffer_name,        { Name of existing buffer }
                      text,               { Text buffer }
                      stream_buf,         { Returns stream ID }
                      status );           { Completion status }

        IF status.all = status_$ok THEN
        BEGIN

            { Read data from existing paste buffer. }

            writeln ( '          =================                ');
            writeln ('This is the contents of paste buffer ',buffer_name, ':');
            writeln;

            WHILE status.all = status_$ok DO
            BEGIN

                { Read a line and write it to screen. }

                stream_$get_rec ( stream_buf,     { Stream ID }
                                  ADDR(info),     { Address of input line }
                                  SIZEOF(info),   { Length of input line }
                                  retptr,         { Returns pointer to input }
                                  retlen,         { Returns length of input }
                                  seek_key,       { Seek key }
                                  status);        { Completion status }
```

Example 5-23. Using Paste Buffers (Cont.)

```
        IF status.code = stream_$end_of_file THEN  { Test for EOF }
        EXIT;
        { Write buffer line to screen }
        writeln ( '    ', retptr^ : retlen );

        IF (status.all <> status_$ok) THEN
        error_routine;
    END;   { While there is input }

END { if }

ELSE IF status.code =  stream_$name_not_found THEN
BEGIN
    { Input data in new paste buffer }

    pbufs_$create ( buffer_name,    { Name of buffer }
                    text,           { Text buffer }
                    stream_buf,     { Returns stream ID of buffer }
                    status );       { Completion status }
    check_status;

    { Get information from keyboard for paste buffer }

    writeln ( '          =================              ');
    writeln;
    writeln (' Type information for paste buffer, one line ');
    writeln (' at a time. Or type CTRL/Z to stop.' );

    WHILE NOT eof DO
    BEGIN      { User has input. }

        readln(info);
        buflen := SIZEOF(info);
        WHILE (info[buflen] = ' ') AND (buflen > 0 ) DO
        buflen := buflen - 1;     { Get rid of trailing blanks }
        buflen := buflen + 1;
        { Terminate line with NEWLINE character: }
        info[buflen] := CHR(pad_$newline);

        stream_$put_rec ( stream_buf,   { Stream ID }
                          ADDR(info),   { Address of input line }
                          buflen,       { Length of input line }
                          seek_key,     { Seek key }
                          status);      { Completion status }
        check_status;

        writeln;
        writeln (' Type another line, or CTRL/Z to stop. ');

    END;  { while not eof }

    writeln ( '          =================              ');
    writeln ('Information is now in the paste buffer: ',buffer_name);
    writeln;
END { else if }
```

Example 5-23. Using Paste Buffers (Cont.)

```
        ELSE WRITELN ( ' Cannot read or write to paste buffer.');
        RESET ( INPUT );  { Reset INPUT to set EOF to TRUE. }

        { Repeat prompt }

        writeln ( '           ==================            ');
        writeln ( ' Type the name of the paste buffer: ');
        writeln ( ' Or type STOP to quit. ');
        readln  ( buffer_name );

        IF ( buffer_name = 'STOP') OR ( buffer_name = 'stop')
        THEN done := TRUE;

    END; { while not done }
END.    { pbufs_paste_buffer }
```

Example 5-23. Using Paste Buffers (Cont.)

5.10. Using the Touchpad Manager

You can control how the system processes the touchpad or mouse input by using system calls with the prefix TPAD. These calls let you

- Control touchpad mode using TPAD_$SET_MODE.
- Inquire about the mode using TPAD_$INQUIRE.
- Re-establish the touchpad raw data range using TPAD_$RE_RANGE.
- Re-origin the touchpad or mouse in relative mode using TPAD_$SET_CURSOR .

In addition to these calls, there are several display driver interface (SMD) calls for using a customer-provided tablet or other locator device. For details on SMD calls, see the *DOMAIN System Call Reference* manual.

You can operate a touchpad or bitpad in absolute mode, relative mode, or absolute/relative mode. The mouse operates only in relative mode. The mode of operation determines how the touchpad corresponds to the display screen. You can change the mode of operation with the TPAD_$SET_MODE call.

You can also affect the operation of the touchpad or mouse by setting the origins, scaling parameters, and the hysteresis factor. All of these are described below.

5.10.1. Absolute Mode

Absolute mode makes the touchpad correspond directly to the absolute point on the screen. That is, whenever you touch the pad, the cursor jumps to the corresponding location on the screen. Moving your finger across the touchpad moves the cursor across the screen in the same direction.

Absolute mode maps the touchpad to a part of the screen dictated by the scaling factor and the origin value.

By default, the origin value is 0,0; so the top left edge of the touchpad represents the cursor positions at the top left edge of the screen. This means that the touchpad maps roughly onto the full screen.

You can change the mode of operation with the TPAD_$SET_MODE call. You can also change the origin value with TPAD_$SET_MODE, so that the touchpad manager sets the origin to a location other than the top left edge of the screen. For details, see Section 5.10.7.

5.10.2. Relative Mode

Relative mode makes the touchpad respond only to finger movement, relative to the current position. That is, it does not respond when the finger first touches the pad, but rather, when it starts moving from the initial point of contact.

You typically use the touchpad in relative mode to *push* the cursor across the screen by rubbing the touchpad. Note that this is the only meaningful mode for a mouse: all mouse movement begins from the current cursor position.

Relative mode is useful when you want the cursor to have a fine resolution in a small area. To get finer resolution, you can call TPAD_$SET_MODE with smaller scale factors. For details, see Section 5.10.4.

When the touchpad or mouse maps to a smaller area of the screen, the user can reach distant areas of the screen by *stroking* the touchpad or mouse. Each stroke moves the cursor closer to the desired area.

You can also change the speed of the cursor movement, so that quick strokes make the cursor move more rapidly. As a result, a quick movement across the pad will move the cursor further than a slow, more deliberate move that covers the same distance.

As the user moves a finger across the touchpad, the pad produces points that are offset from the first point of origin by the distance and direction the finger has moved. For details on the point of origin, see Section 5.10.7.

5.10.3. Absolute/Relative Mode

Absolute/relative mode makes the touchpad respond to the first touch (as in absolute mode), and then in relation to the current position (as in relative mode). In absolute/relative mode, the effect of lifting your finger from the touchpad depends on how long you break contact. If you lift and replace your finger quickly (within half a second) the cursor does not move. But if you lift your finger longer than half a second, the cursor jumps to a new absolute position when you place your finger on the pad again.

Absolute/relative mode is useful for *jumping* the cursor from one place to another, and then carefully positioning it in the new area. For example, this mode is commonly used to move the cursor from one window to another, and then point to a character in the second window.

In absolute/relative mode, the first point the touchpad produces during any use is based on scaling factors that make the touchpad describe the full screen. (For example, x=800, y=1024). Further points are offset from the first point, based on your finger's movement across the pad. The scaling factors you specify in TPAD_$SET_MODE determine how coarse or fine your control is during the relative part of absolute/relative mode.

5.10.4. Changing Touchpad Sensitivity with Scale Factors

The touchpad manager scales the data into raster units. The manager then multiplies scale factors by the prescaled data, to get the raster unit values that the Display Manager understands. You can change scale factors with the call, TPAD_$SET_MODE, to determine how much control the touchpad will have in relative mode. (Scale factors have no meaning in absolute mode.)

The default scale factors map the touchpad to the entire screen. Table 5-4 shows how the x and y factors for the display are divided by the prescaled data, to result in values for x and y in raster units.

Table 5-4. Touchpad Scale Factor Values for Display

X Factor	Y Factor	Display	X Value (Raster Units)	Y Value (Raster Units)
800	1024	Portrait	0 - 799	0 - 1023
1024	800	Landscape	0 - 1023	0 - 799

You can specify smaller scale factors with the TPAD_$SET_MODE call, so that the touchpad maps to a smaller area of the screen. This allows you to make the touchpad or mouse more *finely tuned.*

5.10.5. Timing Factors for the Touchpad or Bitpad in Relative Mode

If you lift your finger from the touchpad for less than one-eighth of a second, the touchpad manager ignores it. If you lift your finger for longer than one-eighth of a second, the touchpad manager automatically re-origins the pad (as if you had called TPAD_$SET_CURSOR) to the last point the pad produced.

If the cursor movement is tied to relative mode, you can make the cursor go to the right by lifting your finger for longer than one-eighth of a second, and touching the pad again on the left edge. By doing so, you re-origin the pad, and make it produce the same data it was producing when you lifted your finger. By repeatedly stroking the touchpad to the right, you keep moving the cursor to the right. Since you can re-origin the touchpad, you typically use relative mode with lower scale factors, to produce more precise cursor control.

In absolute/relative mode, the touchpad manager ignores finger movement that lasts less than one-eighth of a second. If your finger leaves the pad longer, the touchpad manager re-origins the pad to let you put your finger down somewhere else on the pad. If your finger leaves the pad for more than half a second, the touchpad manager concludes that this use of the pad has ended, and the next time you touch the pad will be an absolute point.

5.10.6. Changing the Origin in Absolute Mode with TPAD_$SET_MODE

In absolute mode, the point of origin normally corresponds to the upper left corner of the screen (0,0). You can change the point of origin so that it corresponds to another part of the screen with the TPAD_$SET_MODE call.

This is useful for applications that need to move the cursor within a fixed window rather than the entire screen. For example, your program might display a menu in one window. You can reset the origin of the touchpad so that it resolves to a point in the menu window.

5.10.7. Setting the Origin in Relative Mode with TPAD_$SET_CURSOR

The system "remembers" the last cursor position delivered by a locator device. When a new data point comes from the mouse, or from the touchpad or bitpad in relative mode, a **displacement** is computed and applied to the last locator position. The TPAD_$SET_CURSOR call makes the system *forget* the last locator position, and use the value passed in the call instead. The next locator data will then start from this new position instead of its former position. You will rarely need to make this call, as GPR and the Display Manager make the call at appropriate times.

5.10.8. Hysteresis Factor

The **hysteresis factor** prevents the touchpad manager from responding to any minor movements your finger makes unintentionally. The factor effectively defines a *box* around your finger's current position on the touchpad. The touchpad manger does not move the cursor if your finger stays within the box.

Whenever the touchpad manager senses that your finger has moved from the point last reported, it substracts the hysteresis factor from the absolute value of the change. If the result is zero, or a negative value, the touchpad manager does not move the cursor. If the result is positive, the touchpad manager subtracts the hysteresis factor from the distance moved, and moves the cursor the remaining distance.

You can specify the hysteresis factor with the TPAD_$SET_MODE call. The units of the hysteresis factor refer to screen coordinates. Therefore, the value of the hysteresis factor in terms of physical distance across the screen, depends on the pad's scaling factors. The default hysteresis factor is five.

Chapter 6
Using Eventcounts

The DOMAIN system provides routines to synchronize some events that are external to your program. These events are associated with objects that the system or an external device manages such as:

- A mailbox.
- A stream.
- A peripheral device.
- Graphics input.
- The clock.

To keep track of the above objects, the system increments a number, or eventcount, when its associated event occurs. By using these system-defined eventcounts, a program can wait for events without using computer processing time.

This chapter describes how you can use eventcounts to synchronize external system events. For example, when your program waits for input from a mailbox or a serial I/O line.

You can use another type of eventcount, called a user-defined eventcount, to synchronize activities within your programs. For example, you might want to send data from one program to another, or control access to a file shared by many users. These user-defined eventcounts are described in detail in the *Programming with System Calls for Interprocess Communication* manual.

6.1. EC2 System Calls, Insert Files, and Data Types

To work with eventcounts, use system calls with the prefix EC2. Table 6-1 summarizes the EC2 calls.

To use EC2 calls, you must include the appropriate EC2 insert file for the language in which your program is written. These insert files define constants, data types, and system routines for the EC2 subsystem. The EC2 insert files are:

/SYS/INS/EC2.INS.C	for C.
/SYS/INS/EC2.INS.FTN	for FORTRAN.
/SYS/INS/EC2.INS.PAS	for Pascal.

Most of the EC2 calls described in this chapter require that you specify eventcounts using pointers. For these calls, specify an eventcount using a variable in EC2_$PTR_T format. EC2_$PTR_T is a pointer to an eventcount. In FORTRAN, use the following declaration:

```
INTEGER*2 eventcount
INTEGER*4 ec2_pointer
POINTER /ec2_pointer/ eventcount (1:3)
```

Table 6-1. Summary of EC2 System Calls

Call	Operation
EC2_$READ	Reads the current value of an eventcount.
EC2_$WAIT EC2_$WAIT_SVC	Waits until an eventcount reaches a trigger value.
EC2_$INIT* EC2_$ADVANCE*	Creates and advances user-defined eventcounts.

* Use these calls only when you work with user-defined eventcounts. For more information on these eventcounts, see the *Programming with System Calls for Interprocess Communication* manual.

Some EC2 calls require that you specify an eventcount directly. In these cases, specify a variable in EC2_$EVENTCOUNT_T format. The data type EC2_$EVENTCOUNT_T requires six bytes of storage. In FORTRAN, define this as an array of three INTEGER*2 elements.

This chapter is intended to be a guide for performing certain programming tasks; the data and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference* manual.

6.2. Overview of Eventcounts

When you use eventcounts to synchronize events in a DOMAIN program, you identify the events you want to watch. The system suspends your process, but continues to increment the eventcount until it reaches a trigger value that you also specify. When the eventcount reaches its trigger value, the system wakes your process. Your process then checks for, and responds to the event. (In this sense, the term process means an executing program.)

To use an eventcount in a DOMAIN program, you must specify:

- A pointer to the eventcount associated with the event you are waiting for.
- An eventcount trigger value that, when reached, "triggers" the system to wake your process.

Figure 6-1 shows how the system handles eventcounts during program execution.

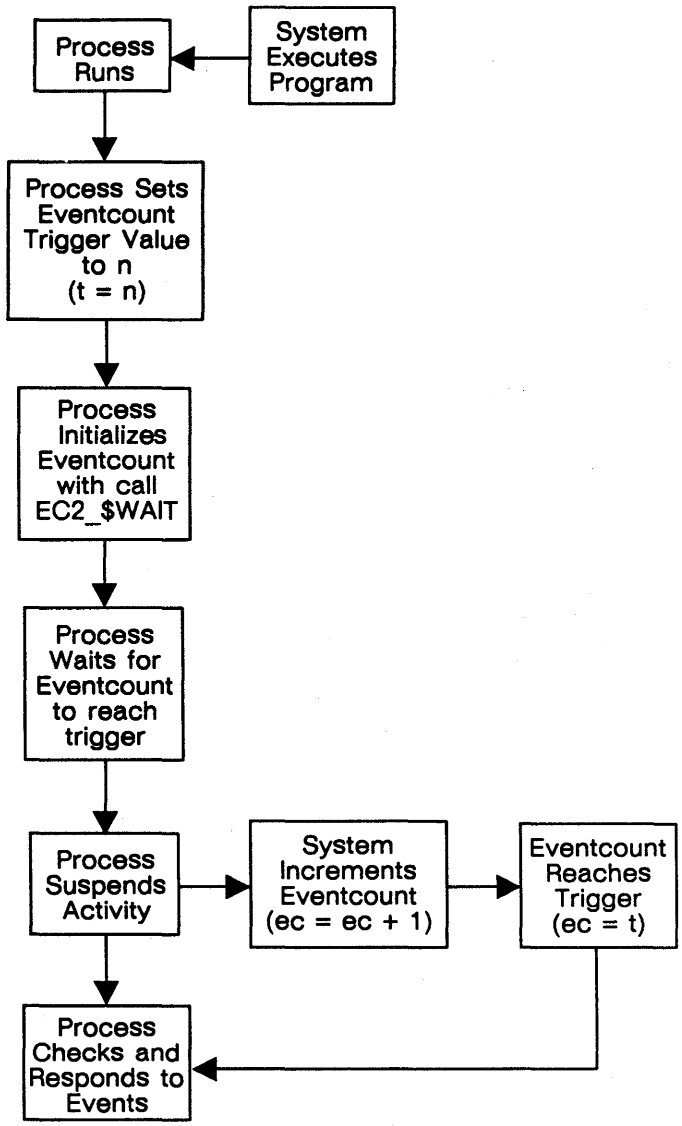


Figure 6-1. Relationship Between a Process and an Eventcount

You can specify several eventcounts to watch for different events, so that the process can respond according to which eventcount reaches its trigger value first.

Note that eventcounts exist in shared memory. Therefore, only programs running on the same node can use the same eventcount.

An alternative (but less efficient) method to wait for events is called busy-waiting. When you use a busy-wait, your program polls for events in a loop. When the event occurs, the program responds to it. A busy-wait is less efficient because it causes the program to monopolize the central processing unit (CPU). This constant use of CPU resources may even delay the events the program is waiting for. Therefore, you should use eventcounts, rather than busy-waits, to wait for events.

6.3. How the System Uses Eventcounts

As stated previously, a system-defined eventcount is one that the system creates and advances. The system automatically creates eventcounts when you:

- Create a mailbox.
- Open a stream.
- Acquire a device.
- Enable graphics input.

It also creates eventcounts for your node's clock. The system uses system-defined eventcounts when managing the associated objects listed above. You can use these eventcounts in your programs, as long as you keep in mind that the system -- not your program -- controls these eventcounts.

The system might not handle eventcounts as you would expect because the system might:

- Advance an eventcount more than once when a single event occurs.
- Advance an eventcount even though the event that the user program is waiting for has not yet occurred.
- Not advance the eventcount for every event that is visible to a user program.

Therefore, your program cannot determine when, or the value by which, the system will advance an eventcount.

To use system-defined eventcounts, a program should use the eventcount as a way to determine when to check for events. After the eventcount wait is satisfied, the program should check to see if the desired event has occurred.

Generally, the best use for system-defined eventcounts is when your program must handle multiple events. That is, when your program is waiting for a number of events, and you want to respond when any of the eventcounts reaches its trigger.

To wait for multiple events, you can use the EC2 calls to create the following cycle:

1. Use the appropriate GET_EC calls to get pointers to the eventcounts.

2. Use EC2_$READ to read the current values of these eventcounts.

3. Establish a loop that uses EC2_$WAIT or EC2_$WAIT_SVC to wait for eventcounts to reach their trigger values.

4. Branch to the code that increments the trigger value and polls for events when an eventcount is satisfied, then return to the wait loop. (Step 3 above.)

The following sections show how to perform each of the above steps to use system-defined eventcounts: Section 6.4 shows how to get and read eventcounts; Section 6.5 shows how to wait for eventcounts; Section 6.6 shows how to respond to events and increment the trigger value; and Section 6.7 shows how to handle asynchronous faults that can occur during this cycle.

Note that each section uses examples from the same sample program. The program waits for two types of events: standard input events (from the input pad) and serial line events. When there is a record in either place, the program gets the record and writes the record to standard output (the transcript pad).

6.4. Getting and Reading Eventcounts

To get pointers for system-defined eventcounts, use any of the GET_EC calls listed in Table 6-2 below.

Table 6-2. EC2 Calls for Obtaining Pointers to Eventcounts

Call	Gets a pointer associated with:
STREAM_$GET_EC	A stream, such as input pad or serial I/O line. Used with stream I/O calls. (Most common.)
MBX_$GET_EC	A mailbox. Used with calls to the mailbox manager.
IPC_$GET_EC	Interprocess communications socket events.
PGM_$GET_EC	A process.
PBU_$GET_EC	A peripheral device. Used when writing GPIO device drivers.
GPR_$GET_EC	Graphics events.
TIME_$GET_EC	The quarter-second clock. (The system increments the time eventcount about every 0.25 seconds.)

When you make your GET_EC calls, place the returned eventcount pointer into an array. The first element in the array is the pointer to the first eventcount, the second element is the pointer to the second eventcount, and so on.

After you obtain pointers to the eventcounts, use EC2_$READ to read the current value of each eventcount into an array of trigger values. In doing so, use the same indexes that you use for your eventcount pointer array. That is, the first element is the value of the first eventcount, the second element is the value of the second eventcount, and so on.

Note that GET_EC and EC2_$WAIT take or return pointers. EC2_$READ uses a dereferenced pointer, as Example 6-1 shows.

NOTE: You must use EC2_$READ to read eventcount values; if you attempt to refer to the eventcount directly, you may obtain an incorrect value, or you may incur a fault such as "odd address error," "access violation," or "reference to illegal address."

Example 6-1 uses STREAM_$GET_EC to get eventcounts for two streams: standard input (usually the keyboard) and a serial input line. The STREAM_$GET_EC calls place the eventcount pointers into an array. Then the example reads the current value of each eventcount into an array of trigger values by dereferencing the pointer to EC2_$READ.

```
PROGRAM sample_use_of_eventcounts;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

CONST          { Define indexes for arrays }
  kbd_ec = 1;  { The first element is for keyboard events.}
  sio_ec = 2;  { The second element is for serial line events.}

VAR
  ec2_ptr : ARRAY [ 1..2 ] OF ec2_$ptr_t; { Array of pointers to two
                    eventcounts. First element points to keyboard EC,
                    second element points to serial line EC. }
  ec2_val : ARRAY [ 1..2 ] OF integer32;  { Array of eventcount trigger
                    values.  First element is trigger for keyboard event;
                    second element is trigger for serial line event. }

  sio_strm : stream_$id_t;              { Stream ID }
  status   : status_$t;                 { Status code }
  seek_key : stream_$sk_t;              { Seek key }

BEGIN    { Main program }

    { Get the standard input eventcount.  Store the ec pointer, returned by the
      call, in the first element of pointer array. }

    stream_$get_ec( stream_$stdin,          { Stream ID }
                    stream_$getrec_ec_key,  { Type of eventcount }
                    ec2_ptr[kbd_ec],        { Returns eventcount pointer }
                    status );               { Completion status }
    check_status;
```

Example 6-1. Getting and Reading System-Defined Eventcounts

```
{ Open a stream to the serial line you'll be reading from
  and get its eventcount.  Store eventcount pointer in the
  second element of pointer array.}

stream_$open( '/dev/sio2',                 { Pathname }
              9,                           { Namelength }
              stream_$write,               { Type of access }
              stream_$no_conc_write,       { Type of concurrency }
              sio_strm,                    { Stream ID returned }
              status );
check_status;

stream_$get_ec( sio_strm,                  { Stream ID }
                stream_$getrec_ec_key,     { Type of eventcount }
                ec2_ptr[sio_ec],           { Eventcount pointer
                                             returned by call }
                status );                  { Completion status }

check_status;

{ Read the current values of each eventcount and store the values
  in the respective elements of the trigger value array. Note that
  you must dereference the pointer to EC2_$READ. }

ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
ec2_val[sio_ec] := ec2_$read( ec2_ptr[sio_ec]^ );
                     .
                     .
                     .
```

Example 6-1. Getting and Reading System-Defined Eventcounts (Cont.)

6.5. Waiting for Events

After creating eventcounts, set up a loop to wait for, and respond to, events. At the beginning of the loop, use either EC2_$WAIT or EC2_$WAIT_SVC to wait for events. The only difference between the calls is in the way they respond to asynchronous faults. See Section 6.7 for more information. The EC2_$WAIT calls have the following format:

ec_satisfied = EC2_$WAIT[_SVC] (ec_plist, ec_vlist, ec_count, status)

Where:

- Ec_satisfied is the number returned by the call, indicating which eventcount is satisfied.
- Ec_list is the array of pointers to the eventcounts you are waiting for.
- Ec_vlist is the array of trigger values for each of the eventcounts. The order of the trigger values must correspond to the order of the eventcount pointers.
- Ec_count is the number of eventcount pointers in the array.
- Status is the status code returned by the call.

When an eventcount in the "ec_list" reaches its trigger value, the EC2_$WAIT call returns an ordinal number, indicating the array subscript of the eventcount that is satisfied. Therefore, a return value of 1 indicates that the first eventcount is satisfied, a return value of 2 indicates that the second eventcount is satisfied, and so on. If more than one eventcount is satisfied, the call returns the one with the smallest subscript.

Branch to the code that responds to the event when the EC2_$WAIT call returns a value. Section 6.6 describes how to respond to the event. After processing the event, return to the top of the loop to wait for more events.

When you first enter the wait loop, use the current eventcount values as your trigger values, as described in Secition 6.4. If you use these trigger values, EC2_$WAIT[_SVC] will indicate that each eventcount is satisfied. By doing this, the program tests for any pre-existing input before waiting for input from each source.

```
NOTE: You usually want to force eventcounts to be satisfied when you
      begin a wait loop. Otherwise, you may miss events that occurred
      before you entered the loop.
```

Example 6-2 uses an EC2_$WAIT loop to wait for two eventcounts. The first eventcount changes when there is new input from the standard input (usually the keyboard); the second eventcount changes when there is new input from a serial line.

If EC2_$WAIT returns a 1, the program branches to the code that gets a record from standard input. If EC2_$WAIT returns a 2, the program branches to the code that gets a record from a serial line. When the program enters the wait loop for the first time, both eventcounts are satisfied. Thus, the first time through the loop, the program tests for any pre-existing input from standard input. The second time through the loop, the program tests for pre-existing input from the serial line. The third time through the loop, the program waits for new input from each source.

```
PROGRAM sample_use_of_eventcounts;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

CONST          { Define indexes for arrays }
  kbd_ec = 1;  { The first element is for keyboard events.}
  sio_ec = 2;  { The second element is for serial line events.}

VAR
  ec2_ptr : ARRAY [ 1..2 ] OF ec2_$ptr_t; { Array of pointers to
                                            two eventcounts }
  ec2_val : ARRAY [ 1..2 ] OF integer32;  { Array of eventcount
                                            trigger values }
  which   : integer;                      { Number returned by
                                            EC2_$WAIT }
  status  : status_$t;                    { Status code }
```

Example 6-2. Waiting for System-Defined Eventcounts

```
BEGIN   { Main Program }

    { Get eventcount pointers for standard and serial line input
      and place pointers into the EC2_PTR array.  Satisfy the
      eventcount by reading the values of each eventcount into
      the EC2_VAL array. }
                        .
                        .
                        .

    { Go into an infinite loop to wait for input from the two sources.
      The first time through, both eventcounts are satisfied. }

    REPEAT
        which := ec2_$wait( ec2_ptr,         { List of pointers }
                            ec2_val,         { List of triggers }
                            2,               { Number of eventcounts }
                            status );
        check_status;

        CASE which OF
            kbd_ec:

                { If WHICH is 1,  handle keyboard events
                  and return to EC2_$WAIT. }
                        .
                        .
                        .

            sio_ec:

                { If WHICH is 2, handle serial input events
                  and return to EC2_$WAIT. }
                        .
                        .
                        .

            END; {case}

    UNTIL FALSE;

END. { sample_use_of_eventcounts }
```

Example 6-2. Waiting for System_Defined Eventcounts (Cont.)

6.6. Responding to Events and Incrementing the Trigger Value

When EC2_$WAIT or EC2_WAIT_SVC returns a value, branch to the code that processes the event. Within this code, you must first increment the trigger value. To increment most triggers, read the current eventcount value and add 1. To increment the time eventcount trigger, read the current eventcount value and add a number of seconds. (The time eventcount gets incremented every 0.25 seconds, so + 4 means + 1 second.)

Next, create an inner loop to poll for and process events. Remember that, although you are responding to an eventcount that is satisfied, the event you are waiting for may not have occurred, so you must check if an event occurred. (In this case, we use the STREAM_$GET_CONDITIONAL system call.) If there is an event, process it and repeat the inner loop. Otherwise, return to the EC2_$WAIT[_SVC] loop.

NOTE: You must increment the trigger value before you check for events. Otherwise, you may return to the EC2_$WAIT[_SVC] loop with a trigger value that is too high. If this occurs, you will continue waiting for the eventcount to increment, even though there is an event you could be processing.

You must use a repeat loop to process all the events, because the program may process many events before reaching the trigger value.

Example 6-3 responds to standard input and serial line input. After incrementing the trigger value, the program uses the system call STREAM_$GET_CONDITIONAL to see whether there is any input. If there is input, the program writes it to the screen. If there is no input, STREAM_$GET_CONDITIONAL returns with a line length of zero, and the program returns to the EC2_$WAIT loop.

```
PROGRAM sample_use_of_eventcounts;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';


CONST           { Define indexes for arrays }
  kbd_ec = 1;   { The first element is for keyboard events.}
  sio_ec = 2;   { The second element is for serial line events.}


VAR
  ec2_ptr : ARRAY [ 1..2 ] OF ec2_$ptr_t; { Array of pointers to
                                            two eventcounts }
  ec2_val : ARRAY [ 1..2 ] OF integer32;  { Array of eventcount
                                            trigger values }
  which    : integer;                     { No of satisfied eventcount }
  sio_strm : stream_$id_t;                { Stream ID }
  status   : status_$t;                   { Status code }
  seek_key : stream_$sk_t;                { Seek key }
  line     : string;                      { Buffer where record
                                            may be read }
  linep    : ^string;                     { Pointer to buffer where line
                                            is read }
  linelen  : integer32;                   { Length of record }
```

Example 6-3. Responding to System-Defined Eventcounts

```
BEGIN

    . { Get eventcount pointers for standard and serial line input
    .   and place pointers into the EC2_PTR array.  Read the
    .   value of each eventcount into the EC2_VAL array. }

    { Go into an infinite loop to wait for input from the two sources.
      The first time through, both eventcounts are satisfied. }

    REPEAT
        which := ec2_$wait( ec2_ptr,           { List of pointers }
                            ec2_val,           { List of triggers }
                            2,                 { Number of eventcounts }
                            status );
        check_status;

        CASE which OF

            kbd_ec:  { If WHICH is 1, enter keyboard loop. }

            BEGIN

                { Read the current eventcount, increment it,
                  and save it as the new trigger. }

                ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ ) + 1;

                { Get and write records.  When there are
                  no more, return to the outer loop. }

                REPEAT
                    stream_$get_conditional( stream_$stdin,  { Stream ID }
                                             ADDR( line ),   { Buffer to
                                                               read line }
                                             SIZEOF ( line ),{ Bufferlen }
                                             linep,          { Pointer to
                                                               returned
                                                               data }
                                             linelen,        { Length of
                                                               data }
                                             seek_key,
                                             status );
                    check_status;

                    IF linelen > 0 THEN
                        writeln( '*KBD* ', linep^:linelen );

                UNTIL linelen = 0;     { No more records to read. }
                END;  { kbd_ec section }
```

Example 6-3. Responding to System-Defined Eventcounts (Cont.)

```
        sio_ec:   { If WHICH is 2, enter serial line loop.}

        BEGIN

            { Read the current eventcount, increment it,
              and save it as the new trigger. }

            ec2_val[sio_ec] := EC2_$read( ec2_ptr[sio_ec]^ ) + 1;

            { Get and write records.  When there are
              no more, return to the outer loop. }

            REPEAT
                stream_$get_conditional( sio_strm,
                                         ADDR( line ),   { Buffer to
                                                           read line }
                                         SIZEOF( line ), { Bufferlen }
                                         linep,          { Pointer to
                                                           returned
                                                           data }
                                         linelen,        { Length of
                                                           data }
                                         seek_key,
                                         status );       { Completion
                                                           status }
                check_status;
                IF linelen > 0 THEN
                    writeln( '*SIO* ', linep^:linelen );

            UNTIL linelen = 0; { No more records to read. }
            END; { sio_ec section }

        END; {case}

    UNTIL FALSE; { Program continues until a CTRL/Q is typed at keyboard. }
END. { sample_use_of_eventcounts }
```

Example 6-3. Responding to System-Defined Eventcounts (Cont.)

6.7. Handling Asynchronous Faults during Eventcount Waits

This section describes what to do when an asynchronous fault occurs during an EC2_$WAIT system call. For a more detailed description of fault handling, see Chapter 2.

When you use EC2_$WAIT or EC2_$WAIT_SVC, you cause a program to wait until the eventcount reaches its trigger value. During that wait, though, an asynchronous fault can occur. An asynchronous fault is a fault generated outside your program, such as when someone types a CTRL/Q sequence at the keyboard to terminate a program.

If a program does not use any fault-handling techniques to handle asynchronous faults, then the system aborts the program when an asynchronous fault occurs. You can use any of these techniques to handle a fault in the following ways:

- Declare a clean-up handler with PFM_$CLEANUP to perform clean-up operations. The clean-up handler aborts normal processing and destroys the program's context, so it cannot return to the place where the fault occurred.

- Declare a fault handler with PFM_$ESTABLISH_FAULT_HANDLER to handle the fault. You can respond to a fault by providing the fault handler with any corrective actions. The fault handler can return to the program where the fault occurred and continue normal processing.

- Disable asynchronous faults with PFM_$INHIBIT. This causes the program to ignore asynchronous faults until you reenable the faults by calling PFM_$ENABLE. At this time, the system reports the first fault (if any) that occurred while faults were inhibited.

You can control your program's response to an asynchronous fault differently, depending on which of the above techniques you use, and whether you use the EC2_$WAIT or EC2_$WAIT_SVC call. Table 6-3 shows how EC2_$WAIT and EC2_$WAIT_SVC respond to an asynchronous fault, if faults are enabled. Table 6-4 shows how EC2_$WAIT and EC2_$WAIT_SVC act when asynchronous faults are disabled.

Table 6-3. Wait Actions When Asynchronous Faults are Enabled

Call	Error-Handling Technique	
	Clean-Up Handler	**Fault Handler**
EC2_$WAIT	Executes clean-up handler.	Executes fault handler. If fault handler returns control to the interrupted code, it continues waiting.
EC2_$WAIT_SVC	Executes clean-up handler.	Executes fault handler. If the fault handler returns control to the interrupted code, it returns the error EC2_$WAIT_QUIT.

Table 6-4. Wait Actions When Asynchronous Faults are Inhibited

Call	Error-Handling Technique	
	Clean-Up Handler	**Fault Handler**
EC2_$WAIT	Defers fault and continues waiting.	Defers fault and continues waiting.
EC2_$WAIT_SVC	Does not handle fault, but returns the error EC2_$QUIT.	Does not handle fault, but returns the error EC2_$WAIT_QUIT.

When you use EC2_$WAIT or EC2_$WAIT_SVC, you need to understand how your program will respond if an asynchronous fault occurs. You must ensure that the program performs any required clean-up actions if a fault occurs.

At times, you want to be sure that your program handles the event it is waiting for without being interrupted. You can do so using either the EC2_$WAIT or the EC2_$WAIT_SVC call. Section 6.7.1 shows how you can inhibit asynchronous faults during EC2_$WAIT calls with the time eventcount. Section 6.7.2 shows how you can inhibit these faults using EC2_$WAIT_SVC.

6.7.1. Disabling Asynchronous Faults with EC2_$WAIT

You might want to disable asynchronous faults to prevent your program from being interrupted during the wait cycle. If you disable faults, you must ensure that your program does not wait indefinitely.

You can disable asynchronous faults using EC2_$WAIT, as long as you know that the wait can be satisfied in a short period of time. To make sure, you can include a time eventcount as your final event. This way, even though your program ignores faults, it continues waiting for only the time specified by the time eventcount. You will want to list the time event last, in case another event gets satisfied at the same time. (If more than one eventcount gets satisfied simultaneously, the call returns the smallest subscript.)

Example 6-4 shows how to disable asynchronous faults. It uses a time eventcount to make sure your program does not wait indefinitely.

```
PROGRAM ec_wait_for_time;

{ This program inhibits asynchronous faults from occuring while waiting for
  input.  If no input occurs within 20 seconds, the time eventcount will
  be satisfied, and the program will enable asynchronous faults. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/ec2.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/pgm.ins.pas';

CONST
  kbd_ec = 1;
  time_ec = 2;

VAR
  ec2_ptr     : array [1..2] of ec2_$ptr_t;
  ec2_val     : array [1..2] of integer32;
  which       : integer;
  status      : status_$t;
  seek_key    : stream_$sk_t;
  line        : string;          {return buffer}
  linep       : ^string;
  linelen     : integer32;
  name        : string;
  time_enough : boolean;

{ ** CHECK_STATUS error reporting procedure ************************** }

BEGIN  { MAIN }
   .
   .
   .  { Get any other eventcounts. }
   { Get a time eventcount to wait an amount of time. }

    time_$get_ec (time_$clockh_key,    { time-key }
                  ec2_ptr[time_ec],    { returned pointer to ec }
                  status);
    check_status;

    { Prime the eventcount trigger values, except the time eventcount. }
    { Immediately advance the time eventcount so that it will not be
      satisfied before other eventcounts get satisfied. }

    ec2_val[time_ec] := ec2_$read (ec2_ptr[time_ec]^) + 80;
    time_enough := FALSE;

    { Disable CTRL/Q sequence while waiting for input or until the
      time limit is reached. }

        pfm_$inhibit;
        writeln(' Faults inhibited. ');
```

Example 6-4. Handling Asynchronous Faults with A Time Eventcount

```
REPEAT  { Until time eventcount satisfied. }

    { Determine which event count reaches satisfaction first. You force
      all eventcounts to be satisfied except time. }

    which := ec2_$wait (ec2_ptr,     { ec pointer array }
                        ec2_val,     { ec value array }
                        2,           { number of ec's }
                        status);
    check_status;

    CASE which OF

        .  { Process other eventcounts.  This code executes if
        .    other eventcounts are satisified.  Asynchronous faults
        .    cannot interrupt processing. }

        time_ec:  { This code executes if the time limit is reached
                    before any other eventcounts get satisfied. }
        BEGIN

            { Immediately advance the satisfaction value -  20 sec.}

            ec2_val[time_ec] := ec2_$read(ec2_ptr[time_ec]^) + 80;
            pfm_$enable;          {OK to interrupt now.}
            time_enough := TRUE;
            writeln ( ' No action for 20 seconds. ');

        END; {time_ec}
    END; {case}
UNTIL time_enough = TRUE;

pfm_$enable;
writeln(' Faults enabled. ');
.
.  { Continue program. }
.
```

Example 6-4. Handling Asynchronous Faults with A Time Eventcount (Cont.)

6.7.2. Disabling Asynchronous Faults with EC2_$WAIT_SVC

The above example uses the EC2_$WAIT call and a time eventcount to disable asynchronous faults for a specified time. Should a fault occur during that time, it will not respond until after the time limit as specified by the time eventcount. The next example uses EC2_$WAIT_SVC to disable asynchronous faults. An advantage to using this call is that, should a fault occur while asynchronous faults are disabled, you will receive the completion status, EC2_$WAIT_QUIT, immediately.

Example 6-5 shows how to disable asynchronous faults using EC2_$WAIT_SVC within a REPEAT loop. If an asynchronous fault occurs during the wait, EC2_$WAIT_SVC returns an error. The program either exits the loop, if a clean-up handler is in effect, or repeats the loop, if a fault handler that returns control to the program is in effect.

Note that this loop responds differently, depending on whether faults were previously disabled. This example assumes that asynchronous faults were *not* disabled before the loop. (Following this example is a description of how this loop responds if they were disabled.)

```
BEGIN   { main program }
    .
    .
    .
    REPEAT  { Until no faults occur }

        pfm_$inhibit;

        .  { Set up the code that you want to protect from
        .    asynchronous faults here. }
        .

        { Use EC2_$WAIT_SVC to receive the error status,
          EC2_$WAIT_QUIT, if an asynchronous fault occurs.}

        ec2_$wait_svc( pointer_list, trigger_list, status);

        IF status.all = status_$ok
            THEN
                 .       { Handle event }
                 .
                 .
            ELSE         { status.all = EC2_$WAIT_QUIT. }

                 .       { Return things to the state before the wait. For
                 .         example, if you opened a serial line, close it. }
                 .

                PFM_$ENABLE;

                 .       { Fault handler or condition takes over here if a
                 .         fault occurs, then returns control to the UNTIL
                 .         condition. }

        { If fault occurred during EC2_$WAIT_SVC, repeat loop and
          try again.  Otherwise, drop through loop and continue.}

    UNTIL status.all <> EC2_$WAIT_QUIT;
    .
    .
    .
```

Example 6-5. Handling Asynchronous Faults with EC2_$WAIT_SVC

If an asynchronous fault occurs and you use a fault handler, the fault handler takes over when PFM_$ENABLE re-enables faults. Thus, the asynchronous fault occurs before the fault handler returns control to the UNTIL condition. Since the completion status is EC2_$WAIT_QUIT, the loop is repeated. The loop will continue to repeat until the process completes without any faults.

You can, however, prevent the fault from occurring before repeating the loop by preventing the fault handler from taking control after the PFM_$ENABLE call. To do so, the above program can disable faults by including an extra PFM_$INHIBIT call *before* entering the REPEAT loop.

When a fault occurs during the loop, the PFM_$ENABLE cannot enable faults because there is an outstanding PFM_$INHIBIT call. So the fault remains disabled, but the completion status returns the completion code, EC2_$WAIT_QUIT. Thus, the loop gets repeated, but the fault does not occur. The loop will continue until the process completes without any faults.

In other words, the system increments an inhibit count at each PFM_$INHIBIT call, and decrements the count at each PFM_$ENABLE call. It transfers control to the fault handler only if the inhibit count is zero. In this case, the PFM_$ENABLE within the loop decrements the count to one. When the loop is repeated, the PFM_$INHIBIT call increments the count again, so the inhibit count never reaches zero within the loop.

Table 6-5 below summarizes how the above program would respond, depending on whether you use a clean-up or fault handler, and whether or not you disable asynchronous faults before entering the loop.

Table 6-5. Program Results if a Fault Occurs During a Wait

Asynchronous Faults Inhibited Before REPEAT loop?	Fault Handler Used?	Results if a fault occurs during the wait.
No	No	The ELSE clause restores items that were set before the wait. When the loop re-enables faults with PFM_$ENABLE, the fault occurs. The clean-up handler handles the fault, and the program exits.
No	Yes	The ELSE clause restores items that were set before the wait. When the loop re-enables faults with PFM_$ENABLE, the fault occurs. The fault handler handles the fault, and returns control to the UNTIL condition. Since STATUS.ALL returns EC2_$WAIT_QUIT, the loop is repeated.
Yes	No	The ELSE clause restores items that were set before the wait. The PFM_$ENABLE call decrements the inhibit count. This does not re-enable faults, because the inhibit count is not zero. Since STATUS.ALL returns EC2_$WAIT_QUIT, the UNTIL condition is FALSE and the loop is repeated.

Chapter 7
Manipulating Time

The DOMAIN system provides a number of system routines to manipulate time. These routines are prefixed with the letters CAL (calendar routines) and TIME (time routines).

This chapter describes the ways the system represents time, how to get the time from the system, and how to manipulate times.

7.1. CAL and TIME System Calls, Insert Files, and Data Types

In order to use CAL and TIME system calls, you must include the appropriate insert files for the language in which your program is written. These insert files define constants, data types, and system routines for the CAL and TIME subsystems.

The CAL insert files are:

/SYS/INS/CAL.INS.C	for C.
/SYS/INS/CAL.INS.FTN	for FORTRAN.
/SYS/INS/CAL.INS.PAS	for Pascal.

The TIME insert files are:

/SYS/INS/TIME.INS.C	for C.
/SYS/INS/TIME.INS.FTN	for FORTRAN.
/SYS/INS/TIME.INS.PAS	for Pascal.

This chapter is intended to be a guide for performing certain programming tasks; the data and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference* manual.

7.2. How the System Represents Time

The DOMAIN system routines use two representations of time: a system-readable representation and a user-readable representation.

The DOMAIN system internally represents time as the number of 4-microsecond units that have elapsed since midnight (00:00) on January 1, 1980, Greenwich Mean Time (a microsecond is a millionth of a second). Time represented in this fashion is referred to as Universal Coordinated Time (UTC). Throughout this chapter it is referred to as UTC.

DOMAIN uses the predefined data type TIME_$CLOCK_T to store internal time values. This data type is a 48-bit integer value. In this chapter, a system routine argument that stores a value in this way will be referred to as being in TIME_$CLOCK_T format. To obtain a local time, an offset must be added to a UTC time, see Section 7.3.2.

In order to manipulate times (add, subtract, compare) using system routines, both absolute times and relative times must be represented as TIME_$CLOCK_T values. However, TIME_$CLOCK_T values are not readily deciphered as time by people.

To permit users to read time, the DOMAIN system also represents time in a six-integer format, in which the six integers represent year, month, day, hours, minutes, and seconds, respectively.

DOMAIN uses the predefined data type CAL_$TIMEDATE_REC_T to store these integer values. It consists of six 2-byte integers. In this chapter, a system routine argument that stores a value in this way will be referred to as being in CAL_$TIMEDATE_REC_T format.

Times stored in CAL_$TIMEDATE_REC_T format must be converted to TIME_$CLOCK_T format before any time manipulation can occur. Conversely, if you wish to print the result of a time manipulation, it must be converted from TIME_$CLOCK_T to CAL_$TIMEDATE_REC_T format. How to convert internal values into readable form, and how to convert readable representations of time into internal values is described in Sections 7.4 and 7.5.

7.3. Getting System Time

To get the current UTC time in TIME_$CLOCK_T format, use the TIME_$CLOCK system routine. This routine returns one argument -- the current UTC value. Note that this UTC value represents Greenwich Mean Time.

7.3.1. Getting Local Time

There are three ways to get the local time. The way you choose will depend on the format you want.

To get the current local time in TIME_$CLOCK_T format, use the CAL_$GET_LOCAL_TIME routine.

You may also compute the current local time from the UTC by using the CAL_$APPLY_LOCAL_OFFSET routine. The offset value is the number of minutes difference between the local timezone and UTC (Greenwich Mean Time). The CAL_$APPLY_LOCAL_OFFSET routine adds the local timezone offset to the UTC value that you pass it. This routine takes one argument that upon input is the UTC, and upon output is the computed local time. Example 7-1 illustrates this computation:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/cal.ins.pas';

VAR
    clock   : time_$clock_t;

BEGIN

    {get the UTC}
    time_$clock (clock);

    {apply offset}
    cal_$apply_local_offset (clock); {in UTC : out UTC + offset}
```

Example 7-1. Getting Local Time Using an Offset

You can also obtain the local time in CAL_$TIMEDATE_REC_T format (year, month, day, etc.), by using the CAL_$DECODE_LOCAL_TIME routine. Example 7-2 obtains the local date and time in CAL_$TIMEDATE_REC_T format, and writes it to the screen (using a VFMT formatting routine -- see Chapter 8.

```
PROGRAM cal_decode_local;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

VAR
    d_clock : cal_$timedate_rec_t;

BEGIN

    {get decoded local time}
    cal_$decode_local_time (d_clock);

    {write it to the screen}
    vfmt_$write10 ('date: %2WD/%2WD/%4WD  time: %2ZWD:%2ZWD:%2ZWD %.',
                   d_clock.month,
                   d_clock.day,
                   d_clock.year,
                   d_clock.hour,
                   d_clock.minute,
                   d_clock.second,
                   0,0,0,0);        {dummy arguments}


END. {program}
```

Example 7-2. Getting Local Time in Readable Format

The output to the screen from this program would appear as follows:

date: 11/16/1984   time: 08:04:34

7.3.2. Timezone Offsets

The time for any given timezone is calculated by adding a timezone offset value to the UTC.

In the previous section, CAL_$APPLY_LOCAL_OFFSET added the local timezone offset to the UTC. You may also remove the local offset from the local time to result in the UTC, by using CAL_$REMOVE_LOCAL_OFFSET. See Example 7-10 for an example of CAL_$REMOVE_LOCAL_OFFSET.

To obtain the local timezone name and the local timezone offset, use the CAL_$GET_INFO routine. This routine returns one argument that contains both the offset and the name. DOMAIN uses the predefined data type CAL_$TIMEZONE_REC_T to store the information in this argument. It consists of a 2-byte integer containing the offset, and a 4-byte character string containing an abbreviation of the timezone name.

The following program segment gets timezone information, and writes the timezone name and offset.

```
VAR
    {declare GET_INFO variables}
    tz_info  : cal_$timezone_rec_t;

BEGIN
    { get local tz info }
    cal_$get_info (tz_info);

    writeln ('timezone ', tz_info.tz_name);
    writeln ('offset ', tz_info.utc_delta);
```

To obtain the offset values for the eight standard U.S. timezones, the Greenwich Mean Time or UTC timezone, use the CAL_$DECODE_ASCII_TZDIF routine. This routine returns both the timezone name and the timezone offset. You can pass this routine a character string containing the timezone name to determine the offset, or you can pass it a character string containing the time difference, in '-|+ hr:min' format, to determine the offset.

The program in Example 7-3 illustrates both ways to use CAL_$DECODE_ASCII_TZDIF. First, it gets the offset using the timezone name, then it gets the offset using the time difference.

```
PROGRAM time_zone (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
    status   : status_$t;

    {TZDIF variables}
    time_zone   : string;          {name/diff}
    tzn_length  : pinteger;        {namelength}
    tz_dif      : integer;         {tz difference}
    tz_name     : cal_$tz_name_t;  {tz name}
```

Example 7-3. Getting Timezone Offset and Name

```
PROCEDURE check_status;  { for error handling }
BEGIN
    IF (status.all <> status_$ok) THEN BEGIN
        error_$print( status );
        pgm_exit;
        END;
END;{check_status}

BEGIN

    {get offset using timezone name }
    writeln ('What time_zone do you want the difference of? ');
    readln (time_zone);
    tzn_length := 4;
    cal_$decode_ascii_tzdif (time_zone,
                             tzn_length,
                             tz_dif,
                             tz_name,
                             status);
    check_status;

    {write timezone offset to screen}
    writeln ('The timezone offset is: ', tz_dif);

    { get timezone offset using time difference}
    writeln ('Input time difference ( [+|-] HR:MIN ) ');
    readln (time_zone);
    tzn_length := 6;
    cal_$decode_ascii_tzdif (time_zone,
                             tzn_length,
                             tz_dif,
                             tz_name,
                             status);
    check_status;

    {write timezone offset to screen}
    writeln ('The time_zone offset is: ', tz_dif);

END.
```

Example 7-3. Getting Timezone Offset and Name (Cont.)

7.4. Converting from System Time to Readable Time

To convert a value in TIME_$CLOCK_T format to a readable integer format, use the CAL_$DECODE_TIME routine. This routine has two arguments; you input the time in TIME_$CLOCK_T format, and it returns the time in CAL_$TIMEDATE_REC_T format.

The program segment in Example 7-4 gets the current time, manipulates it, converts it to a readable format, and writes it to output:

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';

VAR
    clock   : time_$clock_t;        {internal format}
    d_clock : cal_$timedate_rec_t;  {readable format}

BEGIN
    {get local time in TIME_$CLOCK_T format}
    cal_$get_local_time (clock);
              .
              .
              .
    {manipulate the time}
              .
              .
              .
    cal_$decode_time (clock,
                      d_clock);
              .
              .
              .
    {write the time to the screen or file}
```

Example 7-4. Converting from System Format to Readable Format

7.5. Converting from Readable Time to System Time

DOMAIN permits you to input the date and time in ASCII format and convert it to CAL_$TIMEDATE_REC_T format. For the purposes of instruction, this section will describe how to convert ASCII strings into readable format, as well as how to convert readable format into system format.

To convert the ASCII strings to CAL_$TIMEDATE_REC_T format, you must use two system routines, CAL_$DECODE_ASCII_DATE and CAL_$DECODE_ASCII_TIME. As their names suggest, one converts the date and the other converts the time.

The ASCII string you input to CAL_$DECODE_ASCII_DATE must be in the format, *year/month/day*, for example, "85/3/23". The routine takes this string and puts the corresponding integer values into the date half of the CAL_$TIMEDATE_REC_T data type.

The ASCII string you input to CAL_$DECODE_ASCII_TIME must be in the format, *hour:minutes:second* -- in 24-hour format; for example, "17:54:44". The routine takes this string and puts the corresponding integer values into the time half of the CAL_$TIMEDATE_REC_T data type.

Once you have converted the time from ASCII to CAL_$TIMEDATE_REC_T format, you may wish to convert to TIME_$CLOCK_T format. To convert a value in CAL_$TIMEDATE_REC_T format to TIME_$CLOCK_T format, use the CAL_$ENCODE_TIME routine. This routine has two arguments; you input the time in CAL_$TIMEDATE_REC_T format, and it returns the time in TIME_$CLOCK_T format.

The program segment in Example 7-5 does the following:

- Gets ASCII input for the date and time.
- Converts it to CAL_$TIMEDATE_REC_T format.
- Converts CAL_$TIMEDATE_REC_T format to TIME_$CLOCK_T format.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
    status  : status_$t;

    { DATE variables }
    date   : string;    {input date}
    d_len  : pinteger;

    { TIME variables }
    time   : string;    {input time}
    t_len  : pinteger;

    { ENCODE variables }
    c_clock : cal_$timedate_rec_t; {readable time}
    clock   : time_$clock_t;       {internal time}

PROCEDURE check_status;  { for error handling }
BEGIN
    IF (status.all <> status_$ok) THEN
        error_$print( status );
END;{check_status}


BEGIN

   {get the date input}
   writeln ('what date ( yr/month/day )? ');
   readln (date);
   d_len := 8;
   cal_$decode_ascii_date (date,
                           d_len,
                           c_clock.year,  {load year, month, and day}
                           c_clock.month, {directly into first half }
                           c_clock.day,   {of variable}
                           status);
   check_status;
```

Example 7-5. Converting Time from ASCII strings to System Format

```
{get the time input}
writeln ('what time (hr:min:sec - 24 hr format)? ');
readln (time);
t_len := 8;
cal_$decode_ascii_time (time,
                        t_len,
                        c_clock.hour,
                        c_clock.minute,
                        c_clock.second,
                        status);
check_status;

{convert readable format to internal format}
cal_$encode_time (c_clock,
                  clock);
```

Example 7-5. Converting Time from ASCII strings to System Format (Cont.)

7.6. Manipulating Time

DOMAIN provides three system routines with which to manipulate time: CAL_$ADD_CLOCK, CAL_$SUB_CLOCK, and CAL_$CMP_CLOCK. These routines add two time values, subtract two time values, and compare two time values, respectively. All time values that you pass to these routines must be in TIME_$CLOCK_T format.

7.6.1. Relative Time

Up to this point in the chapter, only absolute time values have been discussed. **Absolute time** is a specific point in time, for example, 8:15:23 on 4/8/58. This section discusses relative time. **Relative time** is an amount of time, for example, five minutes. Some time manipulations result in relative time values, while others require relative time values in order to work properly.

DOMAIN provides two system routines to convert relative time values from TIME_$CLOCK_T format into a number of seconds. They differ in the precision of the result; one truncates any fractional portion of the result, the other doesn't.

To convert a relative time value from TIME_$CLOCK_T format to an integer value representing the number of seconds, use the CAL_$CLOCK_TO_SEC routine. CAL_$CLOCK_TO_SEC converts the time into an integer value representing the number of whole seconds -- the fractional portion is truncated. Example 7-7 uses a CAL_$CLOCK_TO_SEC call.

To convert a relative time value from TIME_$CLOCK_T format to a floating point value representing the number of seconds, use the CAL_$FLOAT_CLOCK routine. CAL_FLOAT_CLOCK converts the time into a floating point value that represents the number of seconds, including the fractional portion. Example 7-8 uses a CAL_$FLOAT_CLOCK call.

Remember, the TIME_$CLOCK_T format represents the amount of time in 4-microsecond units. To convert to seconds, the system simply multiplies the number of 4-microsecond units by the number of seconds per 4-microsecond unit (0.0000004 sec.).

DOMAIN also provides a system routine to convert a number of whole seconds (specified as an integer) into TIME_$CLOCK_T format. To convert a number of seconds into a relative time value in TIME_$CLOCK_T format, use the CAL_$SEC_TO_CLOCK routine. Example 7-6 uses a CAL_$SEC_TO_CLOCK call.

7.6.2. Adding Times

DOMAIN provides the system routine CAL_$ADD_CLOCK to add two times. Use CAL_$ADD_CLOCK to do the following:

- Add two relative times to result in a third relative time.
- Add a relative time to an absolute time to result in a new absolute time.

The program in Example 7-6 adds a number of seconds (relative time) to the current local time (absolute time). Remember, to manipulate times they must be in TIME_$CLOCK_T format.

```
PROGRAM cal_add_times (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

VAR
    seconds     : linteger;
    rel_time    : time_$clock_t;
    clock1      : time_$clock_t;
    d_clock     : cal_$timedate_rec_t;

BEGIN

   {input number of seconds to add to time}
   writeln ('How many seconds to add? ');
   readln (seconds);

   {convert number of seconds to internal value}
   cal_$sec_to_clock (seconds,    {# of secs}
                      rel_time);  {internal format}

   {get local time}
   cal_$get_local_time (clock1);

   {add the times}
   cal_$add_clock (clock1,    {in/out}
                   rel_time);

   {get the result in readable form}
   cal_$decode_time (clock1,   {internal format}
                     d_clock); {readable format}
```

Example 7-6. Adding a Relative Time to an Absolute Time

```
   {write it to the screen}
   vfmt_$write5 ('time resulting from add: %2ZWD:%2ZWD:%2ZWD %.',
                 d_clock.hour,
                 d_clock.minute,
                 d_clock.second,
                 0,0);           {dummy arguments}

END.
```

Example 7-6. Adding a Relative Time to an Absolute Time (Cont.)

7.6.3. Subtracting Times

DOMAIN provides the system routine CAL_$SUB_CLOCK to subtract two times. Use CAL_$SUB_CLOCK to do the following:

- Subtract two relative times to result in a third relative time.
- Subtract a relative time from an absolute time to result in a new absolute time.
- Subtract an absolute time from an absolute time to result in a relative time.

CAL_$SUB_CLOCK is a function that returns a Boolean value indicating whether the result of the subtraction is positive or negative. If the Boolean value returns as TRUE, the result of the subtraction is greater than or equal to zero. If the Boolean returns as FALSE, the result is negative and will not be useful.

The program in Example 7-7 subtracts an input absolute time from the current time. It checks the Boolean return value, and prints an error message if the result is negative. Remember, to manipulate times they must be in TIME_$CLOCK_T format.

```
PROGRAM cal_sub_times (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
    status   : status_$t;

    { DATE and TIME , ENCODE, SUB variables}
    date        : string;
    time        : string;
    c_clock     : cal_$timedate_rec_t;
    clock       : time_$clock_t;
    curr_time   : time_$clock_t;
    sub_check   : boolean;
    num_of_sec  : linteger;
```

Example 7-7. Subtracting Two Times

```
PROCEDURE check_status;  {for error_handling}

    BEGIN
    IF status.all <> status_$ok THEN BEGIN
      error_$print (status);
      pgm_$exit;
      END;
    END;

BEGIN  {main}

   {get the input}
   writeln ('Enter date to subtract ( yr/month/day )? ');
   readln (date);

   {convert ASCII string to system readable date}
   cal_$decode_ascii_date (date,
                           8,           {length of date}
                           c_clock.year,
                           c_clock.month,
                           c_clock.day,
                           status);
   check_status;

   {get the input}
   writeln ('Enter time (hr:min:sec -- 24 hr format)? ');
   readln (time);

   {convert ASCII string to system-readable time}
   cal_$decode_ascii_time (time,
                           8,
                           c_clock.hour,
                           c_clock.minute,
                           c_clock.second,
                           status);
   check_status;

   {convert readable format to internal format}
   cal_$encode_time (c_clock,
                     clock);

   {get local time}
   cal_$get_local_time (curr_time);

   {subtract input time from the current time}
   sub_check := cal_$sub_clock (curr_time,
                                clock);
```

Example 7-7. Subtracting Two Times (Cont.)

```
   {convert difference to seconds}
   num_of_sec := cal_$clock_to_sec (curr_time);

   {check if result is negative - print error}
   IF NOT(sub_check)
       THEN writeln ('Subtraction resulted in negative value')
    ELSE
       writeln ('seconds difference ', num_of_sec);

END.
```

Example 7-7. Subtracting Two Times (Cont.)

7.6.4. Comparing Times

DOMAIN provides the system routine CAL_$CMP_CLOCK to compare two times. Use CAL_$CMP_CLOCK to determine which of two times is greater.

You specify the two times, in the format:

```
   return_value = CAL_$CMP_CLOCK(clock1,
                                 clock2)
```

CAL_$CMP_CLOCK is a function that returns an integer value that indicates the result of the compare.

- If the integer returns as 1, clock1 > clock2.
- If the integer returns as 0, clock1 = clock2.
- If the integer returns as -1, clock1 < clock2.

Remember, to manipulate times they must be in TIME_$CLOCK_T format.

Example 7-8 determines which file was modified most recently by reading the modified time attribute of each file and comparing the times. It writes the most recent modification time to output.

```
PROGRAM time_compare (input,output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/type_uids.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  status                    : status_$t;
  pathname1, pathname2      : name_$pname_t;
  namelength1, namelength2  : integer;
```

Example 7-8. Comparing Two File Creation Times

```
{ INQUIRE variables }
  input_mask        : stream_$inquire_mask_t;
  inquiry_type      : stream_$ir_opt;
  attributes        : stream_$ir_rec_t;
  error_mask        : stream_$inquire_mask_t;

{ time variables }
  time1, time2      : time_$clock_t;
  most_recent_time  : time_$clock_t;
  cmp_check         : integer;
  decoded_time      : cal_$timedate_rec_t;

PROCEDURE check_status;
BEGIN
    IF (status.all <> status_$ok) THEN
    BEGIN
      error_$print( status );
      pgm_$exit;
    END;
END;

BEGIN {main}

    { get the first pathname -- calculate its length }
    writeln ('Input the first pathname:');
    readln (pathname1);
    namelength1 := SIZEOF(pathname1);
    WHILE (pathname1[namelength1] = ' ') AND
          (namelength1 > 0) DO
        namelength1 :=  namelength1 - 1;

    { get the second pathname -- calculate its length }
    writeln ('Input the second pathname');
    readln (pathname2);
    namelength2 := sizeof(pathname2);
    WHILE (pathname2[namelength2] = ' ') AND
          (namelength2 > 0) DO
        namelength2 :=  namelength2 - 1;

   { initialize inquire variables }
   input_mask := [stream_$dtm];                { date/time modified          }
   inquiry_type := stream_$name_unconditional; { get by name even if not open }
   attributes.obj_name := pathname1;
   attributes.obj_namlen := namelength1;

   { get date/time modified on pathname1 }
   stream_$inquire (input_mask,
                    inquiry_type,
                    attributes,
                    error_mask,
                    status);
   check_status;
   time1.high := attributes.dtm;
   time1.low  := 0;
```

Example 7-8. Comparing Two File Creation Times (Cont.)

```
    { get date/time modified on pathname2 }
    attributes.obj_name := pathname2;
    attributes.obj_namlen := namelength2;

    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);
    check_status;
    time2.high := attributes.dtm;
    time2.low  := 0;

    { compare times and assign most_recent_time }
    cmp_check := cal_$cmp_clock ( time1,
                                  time2 );



     CASE cmp_check OF
         0 : { times are equal }
         BEGIN
             writeln(pathname1:-1, ' and ', pathname2:-1, ' are the same age');
             most_recent_time := time1;   { could be time2 -- no difference }
         END;

         1 :  { 1 is older than 2 }
         BEGIN
             writeln(pathname1:-1, ' is newer than ', pathname2:-1);
             most_recent_time := time1;
         END;

        -1 :   { 2 is older than 1 }
         BEGIN
             writeln(pathname2:-1, ' is newer than ', pathname1:-1);
             most_recent_time := time2;
         END;
         OTHERWISE writeln('ERROR -- BAD RETURN VALUE FROM CAL_$CMP_CLOCK');
     END;{case}

     { decode most recent dtm }
     cal_$apply_local_offset(most_recent_time);

     cal_$decode_time( most_recent_time,
                       decoded_time);

     write('DATE/TIME MODIFIED: ');
     write(decoded_time.month:1,'/',decoded_time.day:1);
     write('/',(decoded_time.year MOD 100):1);
     write('   ',decoded_time.hour:1,':',decoded_time.minute:1);
     writeln(':',decoded_time.second:1);
END.
```

Example 7-8. Comparing Two File Creation Times (Cont.)

7.7. Suspending Process Execution

Suspending the execution of a process may be useful when attempting to access system resources that are locked. A process can detect that the resource is locked, suspend itself for a short period of time, then retry the operation.

To suspend the execution of a process, use the system routine TIME_$WAIT. You must specify two input parameters. One parameter is the time. This can be either a relative or absolute time. The other parameter is a predefined value indicating whether the time you specified is relative or absolute.

If you specify a relative time, the calling process suspends execution for the specified amount of time. If you specify an absolute time, the calling process suspends execution until the specified time is reached. In either case, the time must be in TIME_$CLOCK_T format.

The program segment in Example 7-9 shows how to suspend process execution for a relative amount of time. The program attempts to read a locked file, and re-attempts the read every five seconds.

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
    status   : status_$t;
    rel_time : time_$clock_t;

BEGIN
              .
              .
              .
    {Attempt a READ from a locked file}
    {Check the status for a file-locked message}
              .
              .
              .
    {If file is locked, WAIT a RELATIVE}
    {amount of time -- 5 seconds}

    {convert # of seconds to internal format}
    cal_$sec_to_clock (5,        {number of seconds}
                       rel_time);

    time_$wait (time_$relative,      {predefined}
                rel_time,            {time to wait}
                status);

    check_status;
              .
              .
              .
```

Example 7-9. Suspending Process Execution for a Relative Time

In some cases, you may wish to suspend process execution until a specific point in time. For example, you may want to invoke a program that prints a reminder at a specific time.

The program segment in Example 7-10 shows how to suspend process execution until an absolute time is reached. The program takes a reminder message and input time from the user, and prints the reminder when the specified time arrives. Note that TIME_$WAIT expects a UTC time, so the program uses CAL_$REMOVE_LOCAL_OFFSET to remove the local time offset, before calling TIME_$WAIT.

```
PROGRAM time_wait_abs (input, output);

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '//bs/latest/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
    status   : status_$t;
    reminder : string;

    { DATE and TIME variables }
    date     : STRING;
    time     : STRING;

    { ENCODE, WAIT variables}
    c_clock     : cal_$timedate_rec_t;
    abs_time    : time_$clock_t;
    curr_time   : time_$clock_t;
    sub_check   : boolean;
    num_of_sec  : linteger;

PROCEDURE check_status;  {for error_handling}

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print (status);
      pgm_$exit;
      END;
    END;

BEGIN

   {input the reminder text}
   writeln ('Input reminder text ');
   readln (reminder);

   {get the input}
   writeln ('When do you wish to be reminded?');
   writeln ('Date: ( yr/month/day )? ');
   readln (date);
```

Example 7-10. Suspending Process Execution Until an Absolute Time

```
  cal_$decode_ascii_date (date,
                          8,          {date length}
                          c_clock.year,
                          c_clock.month,
                          c_clock.day,
                          status);

  check_status;

  {get the input}
  writeln ('Time: (hr:min:sec -- 24 hr format)? ');
  readln (time);
  cal_$decode_ascii_time (time,
                          8,            {time length}
                          c_clock.hour,
                          c_clock.minute,
                          c_clock.second,
                          status);
  check_status;

  {Convert TIMEDATE_REC_T to TIME_$CLOCK}
  cal_$encode_time (c_clock,   {input}
                    abs_time); {result}

  { remove local offset to a time_$clock_t }
  cal_$remove_local_offset (abs_time);

  {WAIT for an ABSOLUTE time}
  time_$wait (time_$absolute,     {predefined}
              abs_time,           {time to wait until}
              status);
  check_status;

  writeln (reminder);

END.
```

Example 7-10. Suspending Process Execution Until an Absolute Time (Cont.)

7.8. Using the Time Eventcount

Eventcounts are discussed in detail in Chapter 6. Read that chapter for a full understanding of eventcount concepts and techniques. This section describes steps that should be taken when specifically using the time eventcount.

TIME_$GET_EC returns an eventcount that is incremented approximately every 0.25 second (it varies slightly with system load). Using TIME_$GET_EC in conjunction with the EC2_$READ and EC2_$WAIT routines permits you to wait for a specific amount of time to elapse. This is useful, for instance, in a case where you are prompting for keyboard input, but will use a default value if no response to the prompt occurs within a certain amount of time.

The program in Example 7-11 prompts the user to input a program name. If the user does not respond, the program prompts two more times, at ten-second intervals.

Note that when the prompting loop is entered, both eventcounts will immediately be satisfied. The loop will immediately be executed twice: once for the keyboard eventcount, once for the time eventcount. Because of this behavior, the prompt count is advanced once when no prompt is output (when the keyboard eventcount is first satisfied); the test for the prompt count is adjusted to take this into account.

Note also that to advance the satisfaction (trigger) value for the EC2_$WAIT routine, you must add a value to the result of the EC2_$READ call. This value represents the number of incrementations you wish to wait before the value is satisfied. Because the time eventcount is incremented every 0.25 second, each four incrementations is equivalent to one second. Thus, adding 40 to the EC2_$READ value tells the system to wait ten seconds.

```
PROGRAM time_wait_or_default;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/ec2.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/error.ins.pas';

CONST
    time_ec = 1; {ec array indices}
    kbd_ec = 2;

VAR
    status  : status_$t;
    ec2_ptr : array [1..2] of ec2_$ptr_t;
    ec2_val : array [1..2] of integer32;
    which   : integer;

    {GET_CONDITIONAL variables}
    seek_key: stream_$sk_t;
    line    : string;         {return buffer}
    linep   : ^string;
    linelen : integer32;

    name    : string;
    p_count : integer;

PROCEDURE check_status;  { for error handling }
BEGIN
    IF (status.all <> status_$ok) THEN
        error_$print( status );
END;{check_status}

BEGIN

    {Get an eventcount to wait on for input from standard in (usually the kbd)}
    stream_$get_ec (stream_$stdin,            {stream ID}
                    stream_$getrec_ec_key,    {stream-key}
                    ec2_ptr[kbd_ec],          {returned pointer to ec}
                    status);
    check_status;
```

Example 7-11. Using a Time Eventcount to Repeat a Prompt

```
  {Get a time eventcount to wait an amount of time}
  time_$get_ec (time_$clockh_key,    {time-key}
                ec2_ptr[time_ec],    {returned pointer to ec}
                status);
  check_status;

{ Prime the eventcount trigger values }
{ Get the current value of both eventcounts }

  ec2_val[kbd_ec] := ec2_$read (ec2_ptr[kbd_ec]^);
  ec2_val[time_ec] := ec2_$read (ec2_ptr[time_ec]^);


{ NOW GO INTO A LOOP PROMPTING FOR INPUT }
  linelen := 0;
  p_count := 0;

  REPEAT
      {determine which eventcount reaches satisfaction first}
      which := ec2_$wait (ec2_ptr,    {ec pointer array}
                          ec2_val,    {ec value array}
                          2,          {number of ec's}
                          status);
      check_status;

      CASE which OF

         kbd_ec:    {if the keyboard ec value is reached first...}

            BEGIN {REPEAT}
              {immediately advance the satisfaction value}
              ec2_val[kbd_ec] := ec2_$read (ec2_ptr[kbd_ec]^) + 1;

              {get keyoard input}
              stream_$get_conditional (stream_$stdin, {stream ID}
                                       addr(line),    {pointer to buffer}
                                       sizeof(line),  {# of bytes requested}
                                       linep,         {returned ptr to buffer}
                                       linelen,       {returned buffer length}
                                       seek_key,
                                       status);
              check_status;
              IF linelen > 0 THEN
                 name := linep^;

            END;

         time_ec:  {if the time ec value is satisfied first...}
              BEGIN
                 {immediately advance the satisfaction value -  10 sec.}
                 ec2_val[time_ec] := ec2_$read (ec2_ptr[time_ec]^) + 40;
```

Example 7-11. Using a Time Eventcount to Repeat a Prompt (Cont.)

```
                {prompt again}
                IF (p_count < 4) THEN
                  writeln ('Input a program name: ')
                ELSE
                  writeln ('The default program name is being used.');
            END; {time manipulation}

        END; {case}

    {advance the prompt count}
    p_count := p_count + 1;
    {repeat until input is received or 3 prompts have occured}
    UNTIL ((linelen > 0) OR (p_count = 5)) ;
                      .
                      .
                      .

END. {program}
```

Example 7-11. Using a Time Eventcount to Repeat a Prompt (Cont.)

Chapter 8
Formatting Variables with VFMT

At various points during program execution, you may find it desirable to transform the data currently (or soon to be) stored in program variables from one format to another. For instance, you may have a variable containing a hexadecimal value and wish to prompt your user with its ASCII equivalent. Or you might want to tabulate results in fixed columns using scientific notation. Or you might need to parse an input line without worrying about whether the user separates the arguments with spaces or semicolons. What you really need is a set of tools for converting data representations between formats.

Certain high-level programming languages (like FORTRAN and C) provide internal facilities for performing many such operations, and you will no doubt prefer to use those when they are available. But the language may not do everything you want (or, if you are using Pascal, much of anything at all!). In that case, you may find the VFMT routines to be useful.

VFMT performs two classes of operations (named using the program's point of view): **encoding** and **decoding**. **Encoding** means taking program-defined variables and producing strings of human-readable text that represent the values of the variables, in a format that you specify. These encoded values are then often written to output for viewing. **Decoding** means taking human-readable text (typically typed by the user), interpreting it in a way that you specify, and storing the apparent data values in program-defined variables. There are routines that perform these operations on data:

- In internal program buffers (VFMT_$ENCODE and VFMT_$DECODE).
- From standard input or to standard output (VFMT_$WRITE and VFMT_$READ).
- From streams (VFMT_$WS and VFMT_$RS).

All function in the same general manner, as described below.

8.1. VFMT System Calls, Insert Files, and Data Types

To use the VFMT formatter, use system calls with the VFMT prefix. In order to use VFMT system calls, you must include the appropriate insert file in your program. This insert file defines constants, data types, and the system routines for the VFMT subsystem. The VFMT insert files are:

/SYS/INS/VFMT.INS.C	for C programs.
/SYS/INS/VFMT.INS.FTN	for FORTRAN programs.
/SYS/INS/VFMT.INS.PAS	for Pascal programs.

This chapter is intended to be a guide for performing certain programming tasks; the data and system call descriptions in it are not necessarily comprehensive. For complete information on the data types and system calls in these insert files, see the *DOMAIN System Call Reference* manual.

8.2. Data Types That Can Be Formatted with VFMT

The VFMT routines mentioned above allow you to format the following kinds of data:

- ASCII characters.
- 2-byte or 4-byte integers interpreted as signed or unsigned integers in octal, decimal, or hexadecimal bases.
- Single- and double-precision reals in floating point and scientific notations.

This includes the following data types for the languages indicated:

Pascal	CHAR, INTEGER, INTEGER16, INTEGER32, BINTEGER, PINTEGER, LINTEGER, REAL, DOUBLE
FORTRAN	CHARACTER, INTEGER*2, INTEGER*4, REAL*4, REAL*8
C	char, short, int, float, double

8.3. Routine Syntax

Each of the VFMT routines has the following general form. For the specific syntax of the routines, see the *DOMAIN System Call Reference* manual.

```
      VFMT_$WRITE2 ('control-string', arg1, arg2)

Call name
Version
Formatting instructions
Arguments for this version
```

Each routine has three versions that differ only in the number of arguments which they accept (and thus the number of variables that they can format at one time): either two, five, or ten. Use the version that best suits your needs, filling any unnecesary arguments with dummy values.

Many formatting instructions in the control string "look" to the arguments later in the calling sequence for information about where to read and write data. The first instruction that needs more information will consume the first argument, the second instruction the second argument, and so on, until the instructions are exhausted. Unused arguments are ignored and must be present only to satisfy the compilers (hence, the dummy arguments mentioned above). The remainder of this chapter shows you how to construct the control string and apply the VFMT routines to common tasks.

8.4. Simple Examples

First, let's look at a few short examples to get the flavor of VFMT. Example 8-1 takes a variable inside a program and writes (encodes) it to standard output, doing some simple conversion along the way.

```
Source:

     i := 65;
     vfmt_$write5 ('%d%m1a%h%.',i,i,i,0,0);

     {Print 'i' in decimal (%d), ascii (%a), and hexadecimal (%h) form.
      Each of these instructions consumes one of the 'i' arguments.
      The 'm 1' indicates that the ascii string has length 1.
      The '%.' indicates the end of the control string and causes a
      newline in the output.
      Zeros fill dummy argument slots since this version requires 5.}

Result:

     65A41
```

The answer is right, but it is pretty hard to read. A few intervening spaces should help.

```
Source:

     i := 65;
     vfmt_$write5 ('%d%7X%m1a%10X%h%.',i,i,i,0,0);
     {'%nX' inserts n blanks in the output.}

Result:

     65       A          41
```

Example 8-1. Writing (Encoding) a Variable to Output using VFMT_$WRITE

As you can see, the control string can appear pretty complicated, even for a simple operation. Example 8-2 is a short decoding example.

```
Source:

     length := 0;
     write ('Enter 5-character ID: ');

     {Read the ID and get its length.}
     vfmt_$read2('%m 5a%.', count, st, id, length);

     {Check for errors.}
     if st.all <> 0 then error_$print (st);

     {Write a header.}
     vfmt_$write2 ('Length   String%.', 0, 0);

     {Echo the test.}
     vfmt_$write2 ('%2x%wd%6x%m5a%.', length, id);

Result:

     Enter 5-character ID: ABCDE
     Length   String
       5      ABCDE
```

Example 8-2. Decoding a Variable using VFMT_$READ

A more complicated decoding example that reads variable-width input fields appears later in this chapter.

8.5. Building Control Strings

VFMT control strings are generally built from literal text strings and special **format directives**. Since they may contain literal text, control strings must always be enclosed in single quotation marks inside the routine's calling sequence, as demonstrated in the examples above.

8.5.1. Format Directive Overview

Every VFMT **format directive** is preceded by the percent sign character %. Each ends with a character indicating the type of the directive. Between the % and the type character, you may specify options that change the directive's effect. You can enter directives in lowercase or uppercase, and place spaces within directives, without changing their effects. For example, the following two control strings are equivalent:

```
'%125m125ZuA'

'%125    m125 Z u A'

'%                 125M                  125zUa'
```

Directives tell VFMT how to behave:

- Numerical format directives force VFMT to consider the next variable argument as a floating point or scientific floating point number, or as an integer octal, decimal, or hexadecimal number.

- The ASCII format directive forces VFMT to consider the next variable argument as an ASCII character string.

  Numerical and ASCII format directives refer to arguments that must appear later in the calling sequence. As VFMT interprets the control string, each time it comes to a numerical or ASCII format directive, it goes to the next variable argument provided with the call and encodes or decodes that argument in the way specified by that directive.

- Miscellaneous format directives produce a field of spaces, insert a newline character into the buffer, and tab to a particular position in the buffer for reading or writing.

- End-of-string directives, %$ and %., mark the end of the control string. If you use "%." in an encoding operation, it also generates a newline character in the output. Otherwise, the two are identical.

8.5.2. Inserting Literal Text

Control strings for encode (write) operations can insert literal text between the directives. VFMT copies this text literally to the destination in the specified position between the encoded items. This is *not* true, however, for decode (read) operations. In this case, the control string can contain only directives.

To make VFMT copy a percent sign to the destination of an encode operation (instead of interpreting it as a directive), specify two percent signs in a row: %%.

8.5.3. Repeating Control Strings

You can make VFMT interpret part of the control string repeatedly by using the %(and %) directives:

```
%repeat-count(                                     %)
             |                                     |
             | portion of control string to be repeated |
```

The **repeat-count** is an integer value between 1 and 65536, indicating the number of times the text between the directives is to be repeated. You *cannot* nest repeat directives.

8.6. Format Directive Usage

There are 15 VFMT format directives. These fall logically into three groups of approximately equal size (Table 8-1). The first group of directives declares the type of data to be formatted. The second group enables special features within the control string itself. The third group applies principally to the format of the output produced. All of these directives can appear intermingled within a single control string, plus literal text if you are encoding.

8.6.1. Formatting ASCII Data: The %A Directive

The %A directive formats ASCII text. You may include a variety of options between the "%" and the "A" to modify the formatting operation, as described in Table 8-2. A bullet in the **E** column of the table means that the option is permissible when encoding. A bullet in the **D** column means that the option is permissible when decoding. If there are differences in the option's behavior between encoding and decoding, those differences are described in the **Function** column.

Table 8-1. Summary of Format Directives

Data-Related	
`% [fw] [M length] [E] [Z]     [K]       [U|L] A`	encode/decode ASCII
`% [fw|fw.dr]      [E] [Z] [J]    [S|P] [W|L] F`	encode/decode floating point
`% [fw|fw.dr]          [Z] [J]    [S|P] [W|L] E`	encode scientific floating
`% [fw]            [E] [Z] [J] [U|S|P] [W|L] O`	encode/decode integer octal
`% [fw]            [E] [Z] [J] [U|S|P] [W|L] D`	encode/decode integer decimal
`% [fw]            [E] [Z] [J] [U|S|P] [W|L] H`	encode/decode integer hex

Control String-Related	
%"..."	declare characters to be used as field delimiters
%$	end control string
%.	end control string, inserting newline character
%n(	begin repeat range
%)	end repeat range

Format-Related	
%%	output a single %
%/	insert new line character
%nT	tab to certain column for read or write
%nX	insert blank

Table 8-2. %A: Format ASCII Data

Usage: `% [fw] [M length] [Z] [U|L] A` (Encode) `% [fw] [M length] [E] [K] [Z] [U|L] A` (Decode)			
Option	**E**	**D**	**Function**
fw	•	•	An integer between 1 and 65536 inclusive, indicating the field width for this item. **Encoding**: If present, write exactly this number of characters to output. If absent, output only nonblank characters, and then stop (unless the Z option is present). **Decoding**: If present, read exactly this number of characters and, then stop (unless some other ending criterion is in force with the E or M options).
M length	•	•	Alternative string length specifier. "Length" is an integer number (from 1 to 65536 inclusive) of characters. If this option is present, it specifies the length of the ASCII text string (passed as the variable argument) to be written or read. If you omit this option, then VFMT looks for a second variable argument immediately following the string argument in the routine's argument list. This variable argument, which can be a 2-byte or a 4-byte integer, specifies the string's length. (You must use a 4-byte integer if its value could be zero.) **Decoding**: The M option is meaningful only if you don't specify a field width ("fw"). "Length" tells VFMT the total number of characters to read, including any delimiters present. (If you also specify E, early termination may take effect.) If you don't use M, VFMT looks at the routine's next argument to determine the string length. In this case, the *original* value of this integer variable tells VFMT how many characters to read. (Early termination may still take effect. If this happens, VFMT changes the value of the integer variable before returning, indicating the number of meaningful characters it stored in the string variable.)
E		•	Specify early termination **(decode only)**. This option forces VFMT to stop reading when it encounters a delimiter (declared with the %"..." directive). Default delimiters are blank and comma. If a field width ("fw") is also specified, VFMT will stop reading when the first of either of the terminating conditions is met.

Table 8-2. %A: Format ASCII Data (cont.)

Usage: % [fw] [M length] [Z] [U|L] A (Encode) % [fw] [M length] [E] [K] [Z] [U|L] A (Decode)			
Option	**E**	**D**	**Function**
K		•	Ignore leading spaces; i.e., spaces that occur to the left of visible text in the input field (**decode only**). If K forces VFMT to skip over leading spaces, they don't cause early termination, even if you specified E.
Z	•	•	Include trailing spaces (spaces that occur to the right of visible text in the input field) in the string variable. Omitting Z makes VFMT ignore trailing spaces. **Encode**: Specifying Z causes trailing blanks in the variable to be written to output. **Decode**: Specifying Z causes trailing blanks in the input to be read into the variable.
L	•	•	Convert all letters read or written to lowercase. Not valid if "U" is specified.
U	•	•	Convert all letters read or written to uppercase. Not valid if "L" is specified.

8.6.2. Formatting Floating Point Data: The %F and %E Directives

The %F and %E directives format floating point data: %F in regular (FORTRAN "F") format, and %E in scientific notation (FORTRAN "E") format. *Note that %E is valid only for encoding (write) operations, while %F is valid for both encoding and decoding.* The various options available are described in Table 8-3. A bullet in the **E** column means that the option is permissible when encoding. A bullet in the **D** column means that the option is permissible when decoding. If there are differences in the option's behavior between encoding and decoding, those differences are described in the **Function** column.

Table 8-3. %F and %E: Format Floating Point Data

Usage:			
`% [fw | fw.dw] [Z] [J] [S|P] [W|L] {F | E}` (Encode) `% [fw] [E] [S] [W|L] F` (Decode)			
Option	**E**	**D**	**Function**
fw	•	•	An integer between 1 and 100 inclusive, indicating the total field width for the number to be read or written, including decimal point, sign, etc. **Encoding**: If the number to be written exceeds the specified field width, a field overflow occurs, and VFMT returns a field filled with asterisks (*). If no field width is specified, VFMT uses as few characters as it can, with two digits after the decimal point. **Decoding**: If no field width is specified, VFMT uses early termination (see the "E" option), and stops reading at the first delimiter that it encounters.
dw	•		An integer specifying that portion of the field width to be occupied by digits to the right of the decimal point (**encode only**). This number should be less than the field width "fw". If "dw" is not specified, the default value is two digits.
E		•	Specify early termination (**decode only**). If you specify E or omit the field width, VFMT reads the input only until it encounters a delimiter (declared with the %"..." directive). Default delimiters are blank and comma. If you specify E and also specify a field width, VFMT reads until it encounters a delimiter or exhausts the input field, whichever comes first.
Z	•		Add zeros (0) to the left of the number to fill the field width specified by "fw" (**encode only**). This option is only valid if "fw" is also specified.
J	•		Left-justify the number within the field whose width you specified (**encode only**). If "J" is not specified, the number is right-justified within the field.

Table 8-3. %F and %E: Format Floating Point Data (cont.)

Usage: % [fw | fw.dw] [Z] [J] [S|P] [W|L] {F | E} (Encode) % [fw] [E] [S] [W|L] F (Decode)			
Option	**E**	**D**	**Function**
S	•	•	Specify that the number to be read or written has a minus sign if it is negative and no sign if it is positive. This is the default setting. Not valid if "P" is specified. **Decode**: This option is redundant since all floating point numbers being read will be signed.
P	•		Specify that the number to be written has a minus sign if it is negative and a plus sign if it is positive (**encode only**). If "P" is not specified, "S" is the default.
W	•	•	Specify single precision. This means data of type *real, single* (Pascal), *REAL*4* (FORTRAN), or *float* (C). If "W" is not specified, "L" is the default.
L	•	•	Specify double precision. This means data of type *double* (Pascal and C), or *REAL*8* (FORTRAN). This is the default setting. Not valid if "W" is specified.

8.6.3. Formatting Integer Data: The %O, %D, and %H Directives

The %O, %D, and %H directives format integer data: %O in octal format, %D in decimal format, and %H in hexadecimal format. The various options available are described in Table 8-4. A bullet in the **E** column means that the option is permissible when encoding. A bullet in the **D** column means that the option is permissible when decoding. If there are differences in the option's behavior between encoding and decoding, those differences are described in the **Function** column.

Table 8-4. %O, %D, and %H: Format Integer Data

Usage:			
`% [fw] [Z] [J] [U|S|P] [W|L] {O | D | H}` (Encode) `% [fw] [E] [U|S] [W|L] {O | D | H}` (Decode)			
Option	**E**	**D**	**Function**
fw	•	•	An integer between 1 and 65536 inclusive, indicating the minimum field width for the number to be read or written. **Encode**: If the number to be written exceeds the specified field width, VFMT expands the field to the size necessary to display the number. If no field width is specified, VFMT uses as few characters as it can to display the number. **Decode**: If no field width is specified, VFMT uses early termination (see the "E" option), and stops reading at the first delimiter that it encounters.
E		•	Specify early termination (**decode only**). If you specify "E" or omit the field width, VFMT reads the input only until it encounters a delimiter (declared with the %"..." directive). Default delimiters are blank and comma. If you specify "E" and also specify a field width, VFMT reads until it encounters a delimiter or exhausts the input field, whichever comes first.
Z	•		Add zeros (0) to the left of the number to fill the field width specified by "fw" (**encode only**). This option is only valid if "fw" is also specified.
J	•		Left-justify the number within the field whose width you specified (**encode only**). If "J" is not specified, the number is right-justified within the field.
U	•	•	Specify that the number to be read or written is unsigned. This is the default setting. **Encode**: "U" causes VFMT to write an unsigned positive integer, even if the original value was negative. (This is usually what you want for octal or hexadecimal encoding.) **Decode**: "U" causes VFMT to ignore any "+" or "-" signs that may appear in the input being read.
S	•	•	Specify that the number to be read or written has a minus sign if it is negative, and no sign if it is positive. Not valid if "U" or "P" is specified. If neither "S" nor "P" is specified, "U" is the default. **Decode**: If a "-" appears in the number being read, VFMT makes the value negative before assigning it to the variable argument.

Table 8-4. %O, %D, and %H: Format Integer Data (cont.)

Usage: % [fw] [Z] [J] [U|S|P] [W|L] {O | D | H} (Encode) % [fw] [E] [U|S] [W|L] {O | D | H} (Decode)			
Option	**E**	**D**	**Function**
P	•		Specify that the number to be written has a minus sign if it is negative, and a plus sign if it is positive (**encode only**). If "P" is not specified, "U" is the default.
W	•	•	Specify that the number being read or written is a word (2-byte) integer. Not valid if "L" is specified. If "W" is not specified, "L" is the default.
L	•	•	Specify that the number being read or written is a longword (4-byte) integer. This is the default setting.

8.6.4. Special Control String Directives

The directives described in this section allow you to control the operation of the various other directives inside a control string. Each directive is valid whether you are using the control string for encoding or decoding.

%"..." **Define early termination delimiters.**

The %" directive lets you redefine the characters (represented by '...') that VFMT uses when you specify the "E" (early termination) option with any directive. VFMT considers the appearance of any of the delimiters you specify as the end of a field. After the first double quotation mark, enter the list of delimiters you want to specify. End the list with a second double quotation mark. To include a double-quote character as one of the delimiters, specify two double-quote characters in a row. Here are some examples; assume these all appeared within control strings:

%"."";" Declare periods, double-quotes, and semicolons to be valid delimiters.

%", " By default, only a comma and a space are legal delimiters for early termination. This sets VFMT back to its normal state.

%$ **Mark the end of the control string.**

%. **Mark the end of the control string.**

For **encoding** operations, "%." also causes VMFT to write a newline character in the output. For **decoding** operations, "%." is identical to "%$".

%n(**Begin repeat loop.**

The "%(" directive marks the beginning of a portion of the control string to be repeated "n" times. N is *required* and must be an integer between 1 and 65536 inclusive. See Section 8.5.3.

%) **End repeat loop.**

The "%)" directive marks the end of a repeat loop.

8.6.5. Format-Related Directives

The directives described in this section allow you to control miscellaneous properties related to the format of the data being read or written. Each directive is valid for encoding; "%T" and "%X" are also valid for decoding.

%% **Write a literal "%" to output.**

The '%%' directive causes VFMT to write a literal percent sign to output. This directive is valid for encoding only.

%/ **Write a newline to output.**

The "%/" directive causes VFMT to write a newline character to output. This makes subsequent text appear on the next line down. This directive is valid for encoding only.

%nT **Tab to column "n" before next operation.**

The "%T" directive causes VFMT to "tab" to column "n" before reading or writing the next piece of data. N is an integer between 1 and 65536 inclusive, with "1" representing the leftmost column. If you do not specify n (i.e., just "%T"), VFMT uses the next available argument in the calling sequence to determine the desired tabbing value.

This directive is valid for both encoding and decoding. If you are encoding, VFMT tabs to the right by depositing spaces in the buffer, or writing spaces to the stream, whichever applies. If you are decoding, VFMT skips to the specified column, without storing intervening characters anywhere.

%nX **Skip "n" spaces before next operation.**

The "%X" directive causes VFMT to skip "n" spaces before reading or writing the next piece of data. N is an integer between 1 and 65536 inclusive. If you do not specify "n" (i.e., just "%X"), VFMT skips one space.

This directive is valid for both encoding and decoding. If you are encoding, VFMT moves to the right by depositing spaces in the buffer, or writing spaces to the stream, whichever applies. If you are decoding, VFMT skips over the specified number of characters without storing them anywhere.

8.7. Examples

This section contains examples showing you how to format variables using VFMT.

8.7.1. Building a Character Table

Example 8-3 uses VFMT to build a table of ASCII characters with their associated decimal and hexadecimal values:

Source Code

```
PROGRAM vfmt_table;

{ This program builds a table of ASCII characters with associated
  decimal and hexedecial values. }

%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%list;

VAR
  i : integer32;
  c : char;

BEGIN  { Main Program }

    { Write out the header and skip a line. }

    vfmt_$write2 ('%3xDecimal%3xASCII%3xHexadecimal%/%.', { Ccontrol string }
                  0,0);                                    { Dummy arguments }

    { Write the decimal, ASCII, and hex values, skipping
      non-printing control characters. }

    FOR i := 0 to 127 DO BEGIN

        IF i <= 32 THEN
            c := ' '
        ELSE c:= chr(i);

        vfmt_$write5 ('%5x%3d%7x%1m 1a%10x%2h%.', { Control string }
                      i,                          { Decimal integer }
                      c,                          { Character value }
                      i,                          { Hex integer}
                      0,0);                       { Dummy arguments }
    END;
END.    { vfmt_table }
```

Example 8-3. Building a Character Table of ASCII Characters

Output from Example 8-3

```
$ table.bin
   Decimal   ASCII   Hexadecimal

       0                  0
       1                  1
       2                  2
       3                  3
       4                  4
               .
               .
               .
      34       "         22
      35       #         23
      36       $         24
      37       %         25
      38       &         26
      39       '         27
      40       (         28
               .
               .
               .
     119       w         77
     120       x         78
     121       y         79
     122       z         7A
     123       {         7B
     124       |         7C
     125       }         7D
     126       ~         7E
     127                 7F
```

8.7.2. Parsing an Input Line

Example 8-4 takes an input line and breaks it into tokens (nonblank strings). It then returns a table containing each token and its character count:

Source Code

```
PROGRAM vfmt_parse;

{ This program tests string parsing routines under "friendly"
  circumstances. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';
```

Example 8-4. Parsing an Input Line

```
CONST
  max_keyword_string_len = 256;

TYPE

  keyword_string_t = array [1..max_keyword_string_len] of char;
  token_ptr_t = ^token_t;
  token_t = record
            text     : array [1..80] of char;
            len      : integer;
            next_one : token_ptr_t;
            end;
  token_list_t = array [1..128] of token_ptr_t;


VAR
  st,
  status       : status_$t;
  n_tokens     : integer;
  in_string    : keyword_string_t;
  i,
  n_fields,
  in_len       : integer;
  retlen       : integer32;
  token_list   : token_list_t;
  key          : stream_$sk_t;
  junk         : keyword_string_t;
  retptr       : ^keyword_string_t;

{ ****************************************************************** }
{ Declare a procedure that takes a string as input, and returns the  }
{ number of tokens in the string and a list of pointers to them.     }
{ Procedure parse_string uses VFMT to peel tokens off a string.      }
{ Tokens within the string must be separated by spaces or commas.    }
{ ****************************************************************** }

PROCEDURE  parse_string (IN  string_of_tokens : keyword_string_t;
                         IN  len              : integer;
                         OUT n_tokens         : integer;
                         OUT token_ptr_list   : token_list_t);
VAR

 temp     : keyword_string_t; { String buffer }
 i,                           { Last decoded position in string }
 j,                           { Pointer list index }
 k,                           { String position index }
 n_fields : integer;          { Number of decoded fields }
 templen  : integer;          { Length off string buffer }

BEGIN  { parse_string }

    n_tokens := 0;
    n_fields := 0;
```

Example 8-4. Parsing an Input Line (Cont.)

```
    { Copy the input string. }

    temp := string_of_tokens;
    templen := len - 1;      { Strip off newline }


    j := 1;
    k := 1;
    i := 0;

    REPEAT

        { Advance position in token pointer list. }

        new (token_ptr_list [j]);

        { Break string into tokens, one field at a time. }
        { Load decoded field into pointer list variables. }
        { Space and comma are both valid delimiters.  }

        i := vfmt_$decode2 ('%em256kzla%.',          { Control string }
                            temp[k],                 { String buffer }
                            templen,                 { Buffer length }
                            n_fields,                { Number of decoded fields }
                            st,                      { Status }
                            token_ptr_list[j]^.text, { Load text }
                            token_ptr_list[j]^.len); { Load length }
        IF (st.all <> 0) THEN
            error_$print_name (st, 'vfmt parse string ',18);

        { Decrement string length by amount decoded. }
        templen := templen - i;

        { Increment string position index by amount decoded. }
        k := k + i;

        { Increment number of tokens. }
        n_tokens := n_tokens + n_fields;

        { Increment pointer list position. }
        j := j + 1;

    UNTIL templen = 0;

END;   { parse_string }

{ ********************************************************************** }

BEGIN  { Main Program }

    in_len := 256;

    REPEAT

        writeln ;
        writeln ('Type string to parse : ');
```

Example 8-4. Parsing an Input Line (Cont.)

```
        { Get the input string. }
        stream_$get_buf (stream_$stdin,    { Standard input stream }
                         ADDR(junk),       { Buffer address }
                         256,
                         retptr,           { Pointer to ret. data }
                         retlen,
                         key,
                         st);
        in_string := retptr^;
        in_len := retlen;

        { Call the procedure to parse the string. }

        parse_string (in_string,       { Input string }
                      in_len,          { Length of string }
                      n_tokens,        { Number of tokens }
                      token_list);     { Pointers to tokens }

        writeln ;
        vfmt_$write2 ('The string you typed was "%a".%.',
                       in_string,
                       in_len-1);
        writeln ;

        { Write the number of tokens.}
        vfmt_$write2 ('The parser returned %wd substring(s): %.',
                       n_tokens,
                       0);
        vfmt_$write2 ('Length   Substring%.', 0, 0);

        FOR i := 1 TO n_tokens DO

        { Write each token and its length}
        vfmt_$write5 ('%3t%wd%10t%a%.',     { Control string }
                      token_list[i]^.len,   { Length to write }
                      token_list[i]^.text,  { Text to write }
                      token_list[i]^.len,   { Length of text }
                      0, 0);

    UNTIL FALSE;

END.  { vfmt_parse }
```

Example 8-4. Parsing an Input Line (Cont.)

Output from Example 8-4

```
$ parse.bin

Type string to parse :
Joe is a genius.

The string you typed was "Joe is a genius.".

The parser returned 4 substring(s):
Length   Substring
  3      joe
  2      is
  1      a
  7      genius.

Type string to parse :
This is enough of this. I'm getting pretty tired.

The string you typed was "This is enough of this. I'm getting pretty tired.".

The parser returned 9 substring(s):
Length   Substring
  4      this
  2      is
  6      enough
  2      of
  5      this.
  3      i'm
  7      getting
  6      pretty
  6      tired.

Type string to parse :       { CTRL/Q to stop the program. }
?(sh) "parse.bin" - process quit (OS/fault handler)
In routine "PFM_$ENABLE" line 363.
$
```

8.7.3. Reading Strings Using a Variety of Formats

Example 8-5 uses the same VFMT_$WRITE routine with a variety of control string options to read an input line. It demonstrates the subtleties of slightly varying combinations of control string options. In particular, it shows how using early termination, defining an early termination delimiter, and including trailing spaces effects the string length returned by VFMT. Note that when early termination (e) is *not* specified the defined delimiter character is treated as just another character.

Source Code

```
PROGRAM vfmt_test_example;

{ This example shows how to use VFMT to read fixed-length strings
  that may include spaces, but are not followed by trailing spaces.}

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/error.ins.pas';

VAR
  xlen,
  ylen,
  count : integer := 0;
  x,
  y     : array [1..5] of char;
  st    : status_$t;

BEGIN

    writeln (' Type 2 character fields, 5 per field, separated by a comma.');

    { Read the two fields.                                        }
    { Set the only delimiter to "," and set the field width to 5. }
    { Use early termination.                                      }

    vfmt_$read5('%",",%m5ea%m5ea%.', { Control string }
                count,               { Number of fields decoded }
                st,                  { Status }
                x,                   { First field }
                xlen,                { Field length }
                y,                   { Second field }
                ylen,                { Field length }
                0);                  { Dummy }

    IF st.all <> 0 THEN
       error_$print (st);

    { Echo the test.}
    { Write headers. }

    vfmt_$write2 ('Control string was m5ea - early termination/delimiter.%.',
                  0,0);
    vfmt_$write2 ('Length   String%.',
                  0,0);

    { Write the two fields. }

    vfmt_$write5 ('%3t%wd%10t%za%.',
                  xlen,
                  x,
                  xlen,
                  0,0);
```

Example 8-5. Reading Strings Using a Variety of VFMT Formats

```
vfmt_$write5 ('%3t%wd%10t%za%/%.',
               ylen,
               y,
               ylen,
               0, 0);

{ Reinitialize field lengths. }
xlen := 0;
ylen := 0;

writeln (' Type characters, 5 per field, separated by a comma.');

{ Read the two fields.                                       }
{ Set the only delimiter to "," and set the field width to 5. }

vfmt_$read5('%",","%m5a%m5a%.',
             count,
             st,
             x,
             xlen,
             y,
             ylen,
             0);

IF st.all <> 0 THEN
   error_$print (st);

{ Echo the test. }
vfmt_$write2 ('Control string was m5a - default/no delimiter. %.',
               0,0);
vfmt_$write2 ('Length   String%.',
               0, 0);
vfmt_$write5 ('%3t%wd%10t%za%.',
               xlen,
               x,
               xlen,
               0,0);
 vfmt_$write5 ('%3t%wd%10t%za%/%.',
               ylen,
               y,
               ylen,
               0,0);

{ Reinitialize field lengths. }
xlen := 0;
ylen := 0;

writeln (' Type characters, 5 per field, separated by a comma.');
```

Example 8-5. Reading Strings Using a Variety of VFMT Formats (Cont.)

```
{ Read the two fields.                                    }
{ Set the only delimiter to "," and set the field width to 5. }
{ Include trailing spaces.                                }
vfmt_$read5('%","%m5za%m5za%.',
             count,
             st,
             x,
             xlen,
             y,
             ylen,
             0);

IF st.all <> 0 THEN
    error_$print (st);

{ Echo the test.}
vfmt_$write2 ('Control string was m5za - trailing spaces/no delimiter..%.',
              0,0);
vfmt_$write2 ('Length   String%.',
              0, 0);
vfmt_$write5 ('%3t%wd%10t%za%.',
              xlen,
              x,
              xlen,
              0, 0);
vfmt_$write5 ('%3t%wd%10t%za%/%.',
              ylen,
              y,
              ylen,
              0,0);

{ Reinitialize the field lengths. }
xlen := 0;
ylen := 0;

writeln (' Type characters, 5 per field, separated by a comma.');

{ Read the two fields.                                    }
{ Set the only delimiter to "," and set the field width to 5. }
{ Use early termination and include trailing spaces       }

vfmt_$read5('%","%m5eza%m5eza%.',
             count,
             st,
             x,
             xlen,
             y,
             ylen,
             0);
IF st.all <> 0 THEN
   error_$print (st);
```

Example 8-5. Reading Strings Using a Variety of VFMT Formats (Cont.)

```
    { Echo the test.}
    vfmt_$write2 ('Control string was m5eza - trailing spaces/delimiter.%.',
                  0,0);
    vfmt_$write2 ('Length   String%.',
                  0, 0);
    vfmt_$write5 ('%3t%wd%10t%za%.',
                  xlen,
                  x,
                  xlen,
                  0,0);
    vfmt_$write5 ('%3t%wd%10t%za%/%.',
                  ylen,
                  y,
                  ylen,
                  0,0);

END.    { vfmt_test_example }
```

Example 8-5. Reading Strings Using a Variety of VFMT Formats (Cont.)

Output from Example 8-5

```
$ vfmt_example.bin

 Type 2 character fields, 5 per field, separated by a comma.
a b  , cd e
Control string was m5ea - early termination/delimiter.
Length   String
  3      a b
  5       cd e

 Type characters, 5 per field, separated by a comma.
a b  , cd e
Control string was m5a - default/no delimiter.
Length   String
  3      a b
  4      , cd

 Type characters, 5 per field, separated by a comma.
a b  , cd e
Control string was m5za - trailing spaces/no delimiter.
Length   String
  5      a b
  5      , cd

 Type characters, 5 per field, separated by a comma.
a b  , cd e
Control string was m5eza - trailing spaces/delimiter.
Length   String
  5      a b
  5       cd e
```

Chapter 9
Accessing DOMAIN Types with IOS Calls

The **Streams** facility allows DOMAIN programs to perform input/output (I/O) on various types of objects. Among the object types that DOMAIN defines is the unstructured ASCII type (UASC), the record type (REC), the serial I/O line descriptor type (SIO), the magtape descriptor type (MT), and the mailbox (MBX) type.

Each object type that the Streams facility supports has an associated **type manager**. A type manager defines the operations that can be performed on its particular object type. To perform I/O, the type managers call more primitive (or device-dependent) managers. However, these lower-level calls are transparent to the application program because the types have the same I/O interface, the IOS manager. This allows programs to use the same language statements or IOS calls to perform I/O regardless of the object it is manipulating. For example, a program can open an object without having to know what kind of object it opening.

Whenever a program performs an I/O operation, the Streams facility recognizes the object type being manipulated and calls its corresponding type manager. The type manager then performs the I/O operation according to its implementation. For example, the UASC type manager uses MS calls to perform an I/O operation on a UASC object while the MBX type manager uses MBX calls to perform an I/O operation on an MBX object.

Type managers support most IOS calls. However, a manager might not support an IOS call that is not meaningful for its type. For example, the SIO manager does not support such IOS calls as IOS_$DELETE, IOS_$TRUNCATE, or IOS_$SEEK.

The Streams facility is comprised of various type managers that users, as well as DOMAIN, can define. For information on writing your own type manager to implement an object type, see the *Using the Open System Toolkit to Extend the Streams Facility* manual.

This chapter describes how to use IOS calls to access the following object types:

- Mailbox
- Serial line descriptor
- Magtape descriptor

Chapter 4 describes how to use the IOS calls and provides details on the UASC and REC object types. Chapter 5 describes how to write programs using pad object types.

9.1. Overview of DOMAIN Object Types

The following list defines many of the object types that DOMAIN supports:

Unstructured ASCII (UASC)
: UASC objects contain text, commands, listings, program source code, or similar information, represented by ASCII code. They are understood by spoolers, text editors and formatters, shells, and language compilers. Although the data in UASC files is not structured into records, the get and put calls consider the NEWLINE character to be a record (line) delimiter, unless you specify the IOS_$NO_REC_BNDRY option. Many programs arbitrarily write binary data to a UASC object. For example, UNIX programs might write binary data to a UASC object since conventional UNIX does not support types. If your program does, you should use the HDRU type instead. Or, you can set the UASC object attribute (IOS_$OF_ASCII) to FALSE, indicating that the object contains something other than ASCII data. The IOS_$OF_ASCII object attribute is initially set to TRUE when you create the UASC object.

Record (REC)
: REC objects contain data that is retrieved in discrete groups, or records. A record boundary marks the end of each record. Get and put calls recognize these record boundaries, so programs can count on getting data from a single record at a time. DOMAIN's REC type implements several record formats. These formats keep track of how to store the data, so the record implementation does not depend on any data. In contrast, UASC objects depend on NEWLINE characters to mark the end of the record. The IOS_$OF_ASCII object attribute is initially set to FALSE when you create a REC object.

Header Undefined (HDRU)
: A HDRU object is similar to a UASC object except that get and put calls ignore NEWLINE characters. If the HDRU object is created with the IOS_$NO_REC_BNDRY option (as it is in the UNIX read call), then the HDRU object is almost completely identical to a UASC object. The only difference is that the initial value of the IOS_$OF_ASCII object is set to FALSE when you create a HDRU object. You can set this attribute to TRUE to indicate that the object contains ASCII data. (Still, many programs that work with UASC objects will give unexpected results because get and put calls do not recognize the NEWLINE characters as record delimiters.)

Object or Binary (OBJ)
: Binary objects generally contain executable code or program data. They are interpreted only by the processor or another program.

Directory (DIR)
: A directory is a system object that keeps track of related objects.

Input and Transcript Pad (IPAD, PAD)
: Pads are special disk files that contain text and graphics, which users view through windows on the screen. An input pad (IPAD) object accepts user's input from the keyboard and transfers the input to the program line by line. A transcript pad (PAD) object contains a record of the program's dialogue with the user. The program writes its output to the transcript pad after reading input from its input pad.

Mailbox (MBX) A mailbox is an object that two programs use to exchange information. To read or write using mailboxes, you generally use the MBX system calls, described in detail in the mailbox chapter of the *Programming With System Calls for Interprocess Communication* manual. You can also access mailboxes using IOS calls.

Serial Line Descriptor (SIO)
A serial line descriptor object is the way in which a program communicates with another device via a serial port. (Each node has a number of serial ports to which a serial line can be physically attached to connect the node and a peripheral device.) A program must open a stream to the serial port by specifying the name of a predefined SIO descriptor object, set the serial line's characteristics, and call the IOS manager to perform I/O across the serial line.

Magnetic Tape Descriptor (MT)
A magnetic tape descriptor object is the way in which a program communicates with a magnetic tape device. To read to or write from an object on magnetic tape, the program first creates a magnetic tape descriptor object that establishes the volume and object attributes for the magnetic tape. It then calls the IOS manager to perform I/O to and from objects on the tape.

9.2. Accessing Mailboxes

The IOS manager allows you to access mailboxes created with the MBX interface. This feature is useful if you want to write a program that performs I/O independent of whether the object is a mailbox or another type of object.

This section assumes some knowledge of the mailbox (MBX) system calls. For details, see the mailbox chapter of *Programming with System Calls for Interprocess Communication*.

The following is a brief review of the MBX interface:

- The MBX interface is asymmetric. There are two distinct sides to a conversation -- the client side and the server side. While some MBX calls are available and useful to both sides, many of the MBX routines are either client-specific or server-specific.

- The MBX server always creates and initializes the MBX object using the MBX_$CREATE_SERVER call. Once this call is made, the MBX object is "open for business" and clients can make connections to the server through it.

- MBX clients initiate connections by calling MBX_$OPEN, which identifies a specific MBX object. The server of the specified MBX object is notified of this client's desire to connect, and the server then *accepts* or *rejects* the client's open request. In either case, the client waits in the MBX_$OPEN call until the server responds to the open request.

- After the server has accepted the client's open request, the two parties can exchange data until the client closes the channel or the server deallocates it.

Once you understand the MBX interface, it can be useful to know that the *client* side of an MBX session can be written completely with IOS calls, rather than MBX calls. You can use most IOS calls on an MBX object. (However the MBX type manager does not support some IOS calls such as IOS_$SEEK, IOS_$DELETE, or IOS_$TRUNCATE.)

The following sections describe how to write a client using IOS calls.

Note: Only MBX *clients* can access mailboxes through the IOS manager. MBX *servers* cannot use the IOS manager to access mailboxes.

9.2.1. Opening a Mailbox with IOS_$OPEN

To open a mailbox with the IOS manager, call IOS_$OPEN specifying the name of the mailbox in the pathname parameter. If the MBX server accepts the open request, the IOS_$OPEN call succeeds and all subsequent I/O will be over the MBX channel. If the server rejects the open request (or if no server currently controls the MBX object), the IOS_$OPEN call will fail.

Calling IOS_$OPEN is equivalent to calling MBX_$OPEN with one exception: MBX_$OPEN normally allows the client to specify a block of data that should be sent along with the open request. The server evaluates this data before it accepts or rejects the open request. When the open is triggered by a call to IOS_$OPEN, no data accompanies the open request.

9.2.2. Performing I/O on Mailboxes with IOS Calls

To understand how to write to and read from mailboxes you must know how data is stored in a mailbox. Mailboxes have two kinds of data messages: data and partial-data. You can send and receive any number of partial-data messages as long as the sequence terminates with a data message. A **mailbox record** is any number of partial-data messages followed by a data message. Examples of mailbox records are the following:

- data
- partial-data data
- partial-data partial-data partial-data data

To send partial-data messages to the server, use IOS_$PUT with the IOS_$PARITAL_RECORD_OPT. This is equivalent to using the MBX_$PUT_CHR call.

To send complete mailbox records to the server, use IOS_$PUT without the partial record option. This call is equivalent to the MBX_$PUT_REC call. IOS_$PUT (without the partial record option) causes the client to send as many MBX messages as are necessary to contain the supplied data. That is, the MBX-server process may see several MBX messages as a result of a single IOS_$PUT -- every MBX message but the last will be stamped as partial-data.

If the server's channel is full when you try to send a mailbox message, the program suspends until there is room to accept a message. You can specify the IOS_$COND_OPT put option, if you want the call to return immediately. It will return with the IOS_$PUT_CONDITIONAL_FAILED status code. This is equivalent to the MBX_$PUT_xxx_COND call (where xxx is either CHR or REC).

After sending a message to a server's mailbox, you usually want a response. To get a response use one of the IOS get calls, IOS_$GET or IOS_$LOCATE.

The get call attempts to return an entire mailbox record, regardless of how many partial-data messages must be concatenated to form it. If the supplied buffer is not large enough to hold an entire mailbox record, the call returns enough data to fill the requested size and the error, IOS_$BUFFER_SIZE_TOO_SMALL. You can inquire about the number of bytes that remain to be read in the current record by calling IOS_$INQ_REC_REMAINDER. This call returns the "best guess" as to the remaining length of the mailbox record because the entire MBX record may not yet be visible to the MBX client.

If the server's response is not immediately available, you can either suspend the client process until the server's response arrives, or you can have the call return immediately. By specifying IOS_$COND_OPT on the get call, the call returns immediately regardless of whether the server sent a response. If the server did not return the response, the get call returns the IOS_$CONDITIONAL_FAILED error status code.

9.2.3. Example of Accessing a Mailbox with IOS Calls

Note that before you can execute a client MBX program, you must execute a MBX server program to create the mailbox and handle the messages. The /domain_examples directory contains a server program that can handle messages from the client program described in the following program, Example 9-1.

The program in Example 9-1 does the following:

- Opens a connection to a mailbox by calling IOS_$OPEN specifying the name of a mailbox and write access.
- Gets the message from the user using IOS_$GET.
- Puts the message in the mailbox using IOS_$PUT.
- Waits for a response from the server using IOS_$LOCATE. (Since the program doesn't specify the IOS_$COND_OPT, it suspends until the server sends a message.)
- Closes the stream to the mailbox when the user is done by calling IOS_$CLOSE.

```
PROGRAM ios_mbx_client;


%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/mbx.ins.pas';
```

Example 9-1. Writing to and Reading from a Mailbox

```
CONST
  mbx_name    = 'test_mailbox';    { Mailbox name }
  mbx_namelen =  SIZEOF(mbx_name); { Length of mailbox name }
  data_size   =  256;              { Size of input buffer }

VAR

  status     : status_$t;
  stream_id  : ios_$id_t;
  buffer     : string;
  buffer_ptr : ^string;
  ret_length : integer32;
  i          : integer;
  stop       : boolean;

PROCEDURE check_status;  { for error handling }
BEGIN
    IF (status.all <> status_$ok) THEN
        error_$print( status );
END;{check_status}

BEGIN {main}

    writeln;
    writeln (' This program prompts you for a message to send to the server.');
    writeln (' If the server gets the message, it returns the message, ');
    writeln ('                 ''Message Written.'' ');
    writeln;

    { Open the mailbox. }

    stream_id :=  ios_$open (mbx_name,
                             mbx_namelen,
                             [ios_$write_opt,           { write }
                              ios_$unregulated_opt],    { unregulated }
                             status);
    check_status;

    { Read data from keyboard and put it in the mailbox. }

    writeln ('Enter a message for the mailbox or ''q'' to quit.');

    { Get message from keyboard. }
    stop := FALSE;

    ret_length := ios_$get (ios_$stdin,
                            [],
                            buffer,
                            data_size,
                            status);
    check_status;
    IF ((buffer[1] = 'q') OR (buffer[1] = 'Q') AND (ret_length = 1))
         THEN stop := TRUE;
```

Example 9-1. Writing to and Reading from a Mailbox (Cont.)

```
    WHILE NOT stop DO
    BEGIN

{ Put message in mailbox. }

         ios_$put ( stream_id,
                    [],
                    buffer,
                    ret_length,
                    status);
         check_status;

{ Get response from server. }

         ret_length := ios_$locate (stream_id,
                                    [],
                                    buffer_ptr,
                                    data_size,
                                    status);
         check_status;

{ Write message to stdout. }

         ios_$put ( ios_$stdout,
                    [],
                    buffer_ptr^,
                    ret_length,
                    status);
         check_status;
         writeln;
         writeln ( 'Enter a new message; or ''q'' to quit. ');

        { Get message from keyboard. }

        ret_length := ios_$get (ios_$stdin,
                                [],
                                buffer,
                                data_size,
                                status);
        check_status;

         IF ((buffer[1] = 'q') OR (buffer[1] = 'Q') AND (ret_length = 1))
              THEN stop := TRUE;

    END; { while not stop }

{ Close the channel. }

    ios_$close( stream_id,
                status);
    check_status;
END.  { ios_mbx_client }
```

Example 9-1. Writing to and Reading from a Mailbox (Concluded)

9.3. Accessing Serial Lines

Programs can communicate with peripheral devices (such as printers and dumb terminals) across a serial line by using the RS-232 protocol standard. Each node has a number of ports to which a serial line can be physically attached, thereby connecting the node and a peripheral device. To communicate with another device across a serial line, a program must:

- Open a stream to the serial port by opening the SIO descriptor object
- Set attributes for the serial line
- Use IOS calls to perform input and output on streams open to serial lines

9.3.1. Opening a Stream to a Serial Line

To open a stream to a serial line call IOS_$OPEN, specifying the pathname of an SIO descriptor object. The descriptor object is the object that the operating system uses to access the hardware. Table 9-1 lists the predefined names of SIO descriptor objects for every DOMAIN node:

Table 9-1. Default SIO Descriptor Objects Pathnames

SIO Descriptor Object	Serial Line Number
/dev/sio1	serial port 1
/dev/sio2	serial port 2
/dev/sio3	serial port 3
/dev/sio*	Default port (port 1)

* The /dev/sio is the SIO descriptor object of a terminal from which a DSP server is booted.

You can copy and rename SIO descriptor objects without losing their special attributes. However, the objects must be located in the /dev directory so that IOS_$OPEN can open them.

All copies of an SIO object are equivalent for the purposes of concurrency control. If two processes want to share the same SIO line, they must specify IOS_$UNREGULATED_OPT in their IOS_$OPEN calls. However, multiple users within the same process share the same SIO line, regardless of the concurrency control.

Note that you can only connect to an SIO line from a node that is physically connected to the particular line; you cannot connect to SIO lines from remote nodes.

9.3.2. Setting Serial Line Object Characteristics

SIO line objects have a number of attributes that control how the SIO manager interprets data transfers. These attributes control such things as the baud rate (speed) of the serial line, whether characters are echoed as output, and whether the modem is hung up when the SIO line closes. The attributes also define how the SIO manager interprets numerous special characters. For example, by default, the SIO manager interpets CTRL/Z as the "end-of-file (EOF)" character and CTRL/X as the "delete-to-end-of-line" character. The SIO section of the *DOMAIN System Call Reference* manual lists these attributes along with descriptions and default values.

After opening a stream to an SIO line, your program might need to set attributes for the line. To do so, the program would:

- Call SIO_$INQUIRE to determine the current serial line attributes.
- Call SIO_$CONTROL to change the current serial line attributes.

The SIO attributes that you inquire about and set are in SIO_$OPT_T format. You specify the attribute that you want to inquire about or set in the second parameter. SIO_$INQUIRE returns the value (in SIO_$VALUE_T format) of the attribute in the third parameter. If you want to change the value of the attribute, use SIO_$CONTROL specifying a new value in the third parameter.

The value of the third parameter depends on the attribute you specify in the second parameter; they are listed in the description of each attribute in the *DOMAIN System Call Reference* manual. Since these values can be in a variety of forms, the SIO_$VALUE_T is a variant record that can be one of four values: character, Boolean, integer, or a set of enabled errors. To assign a value, you must specify the appropriate field: Specify *b* for attributes that take a Boolean value; *c* for character values, *i* for integer values, and *es* for a set of enabled errors. (For a program example, see Section 9.3.4.)

Note that you must make a separate call for each attribute you want to inquire about or change.

9.3.3. Performing I/O across a Serial Line

After opening and setting the attributes of a serial line, the program can then use the standard IOS calls to send and receive data across the serial line. The program could make the following calls:

- IOS_$PUT to send data to a device.
- IOS_$GET or IOS_$LOCATE to receive data from a device. (In this case, IOS_$LOCATE is no more efficient than IOS_$GET. Since the SIO manager does not support internal buffering, IOS_$LOCATE cannot locate the data and return a pointer to it. Instead, it creates a buffer and calls IOS_$GET to get the data.)
- IOS_$CLOSE to close the stream after completing the data transfer.

To interpret data sent across a serial line, you must use the RS-232 protocol. However, a description of using this protocol is beyond the scope of this manual. Consult the RS-232 standard for this information.

9.3.4. Example of Accessing an SIO Line

The program in Example 9-2 does the following:

- Opens a stream to an SIO line using IOS_$OPEN.

- Inquires about whether the HOST_SYNCH mode attribute is on or off using SIO_$INQUIRE. In HOST_SYNCH mode, the node sends XOFF (CTRL/S) when its input buffer begins to fill, and XON (CTRL/Q) when its input buffer begins to empty again. This allows programs to synchronize high-speed data transfer from computer to computer.

- Assigns the value of the HOST_SYNCH mode attribute. Since this attribute takes a Boolean value, it assigns a value by specifying the Boolean field (.b) of the variant record in SIO_$VALUE_T format.

- Changes the HOST_SYNCH mode to FALSE (if it is TRUE) using SIO_$CONTROL.

- Closes the stream to the SIO line using IOS_$CLOSE.

```
PROGRAM ios_sio_access;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/sio.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/vfmt.ins.pas';

VAR
  {$OPEN variables}
  status      : status_$t;
  pathname    : name_$pname_t;
  namelength  : integer;
  count       : integer;
  stream_id   : ios_$id_t;

  {SIO_$ variables}
  value   :  sio_$value_t;

PROCEDURE check_status;  { for error handling }
 .
 .
 .

BEGIN  {main}

{ Ask user for pathname and convert it to internal format using VFMT. }

    writeln (' Input the pathname of an SIO line:');
    namelength := SIZEOF(pathname);
```

Example 9-2. Accessing a Serial Line

```
    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);
    check_status;

    stream_id := ios_$open (pathname,
                            namelength,
                            [ios_$write_opt],  { Write access }
                            status);  { Regulated concurrency }
    check_status;

    { INQUIRE host-synch }
    sio_$inquire (stream_id,
                  sio_$host_sync, { Inquired option }
                  value,          { Returned value }
                  status);
    check_status;

    writeln (' The host_synch value is: ',value.b);

    IF (value.b = TRUE) THEN BEGIN

        value.b := FALSE;       { Turn off host-synch }

        sio_$control (stream_id,
                      sio_$host_sync,
                      value,
                      status);
        check_status;

        { INQUIRE new host-synch }
        sio_$inquire (stream_id,
                      sio_$host_sync, { Inquired option }
                      value,          { Returned value }
                      status);

        IF status.all <> status_$ok THEN
            ERROR_$PRINT (status);

        writeln (' The host_synch value has been changed to: ',value.b);

    END; {if}

{ Close the channel. }

    ios_$close( stream_id,
                status);
    check_status;

END.
```

Example 9-2. Accessing a Serial Line (Concluded)

9.4. Accessing Files on Magnetic Tape

You can access files that reside on magnetic tapes by using the IOS calls in conjunction with the MTS (Magtape Stream) calls. You access the magtape by first creating and editing a **magtape descriptor object** (MT) which establishes the volume and object attributes for a given magnetic tape.

To create the descriptor object, you use MTS calls. Once you have prepared the magtape descriptor object for the tape you want to access, you can then make IOS calls to read to or write from files on the tape. (Since we traditionally think of magnetic tapes as containing files indicated by a file sequence number, this section refers to objects on a tape as files.)

Before your program can make IOS calls to files on a magnetic tape, a magtape descriptor object for the tape must exist. Once you have created a descriptor object, you can:

- Use MTS calls to change volume and object attributes of the magtape descriptor object.

- Use IOS calls to read from and write to files on the magtape.

When accessing a magtape, you can use most of the IOS calls. (However the MT type manager does not support some IOS calls such as IOS_$SEEK, IOS_$DELETE, or IOS_$TRUNCATE.) Only one process at a time can read from and write to a magtape. The magnetic tape is accessible only to programs executing on the node to which the tape is physically attached.

The following sections describe how to:

- Create and open a magtape descriptor object

- Set attributes of a magtape descriptor object

- Close the magtape descriptor object

- Use IOS calls to perform input and output on magtape files

9.4.1. Creating and Opening a Magtape Descriptor Object

To create a magtape descriptor object for a given magnetic tape, call MTS_$CREATE_DEFAULT_DESC specifying the name and namelength of the descriptor object. Programs can create a magtape descriptor object in any directory.

The descriptor object holds information that the IOS manager uses to open, read, and write files on the tape. For example, the file sequence number attribute indicates which file on the tape the IOS manager is currently operating on. The MTS Data Types section in the *DOMAIN System Call Reference* manual lists the attributes you can control in MTS_$ATTR_T format.

To open a magtape descriptor object, call MTS_$OPEN_DESC specifying the pathname of an existing magtape object, the length of the pathname, and the read-write access, in MTS_$RW_T format. Read-write access indicates whether you want to open the descriptor object for reading or writing. Specify one of the predefined values, MTS_$READ (for read-only access) or MTS_$WRITE (for read and write access). MTS_$OPEN_DESC returns a pointer to the opened object, in MTS_$HANDLE_T format.

Note that a magnetic tape descriptor object must be *open* to read and change the attributes of a descriptor object. However, the object must be *closed* before any IOS calls can operate on the magnetic tape itself.

You can also use MTS_$COPY_DESC to *create* a descriptor object. MTS_$COPY_DESC copies a source magtape descriptor object to a destination magtape descriptor object, opens the destination object, and returns a pointer to it.

9.4.2. Reading and Changing Magtape Descriptor Attributes

Once you have created a descriptor object, you may want to change some of the volume and file attributes. For example, to specify which file on the tape you want to write to, you must specify the appropriate file sequence number.

To change the volume and file attributes of a magtape descriptor object you can do the following:

- Call MTS_$GET_ATTR to determine the current attributes.

- Call MTS_$SET_ATTR to change any attributes.

The attributes you inquire about are in MTS_$ATTR_T format. You specify the attribute that you want to inquire about or set in the second parameter of the call. MTS_$GET_ATTR returns the value (in MTS_$ATTR_VALUE_T format) of the attribute in the third parameter. If you're changing the specified attribute with MTS_$SET_ATTR, you specify the new value in the third parameter.

The value of the third parameter depends on the attribute you specify in the second parameter; they are listed in the description of each attribute in the *DOMAIN System Call Reference* manual. Since these values can be in a variety of forms, the MTS_$ATTR_VALUE_T is a variant record that can be one of three values: integer, Boolean or character. To assign a value, you must specify the appropriate field: Specify *i* for attributes that take integer values, *b* for Boolean values; *c* for character values. (For a program example, see Section 9.4.4.)

Note that you must make a separate call for each volume or file attribute that you want to inquire about or change.

You can also edit magtape descriptor objects interactively with the DOMAIN command EDMTDESC. See the *DOMAIN System Command Reference* manual for details.

9.4.3. Closing a Magtape Descriptor Object

Before you can perform IOS operations on a magtape, you must close the descriptor object. To close a magtape descriptor object, call MTS_$CLOSE_DESC, specifying a pointer to the descriptor object and a value of TRUE or FALSE in the second (update) parameter. The update parameter indicates whether you want the descriptor object to reflect changes you made to the object attributes with MTS_$SET_ATTR. If the value is TRUE, MTS_$CLOSE_DESC makes the changes. If the value is FALSE, MTS_$CLOSE_DESC closes the descriptor object but does not update the attributes.

9.4.4. Example of Writing to a Magtape File

Once you close the magtape descriptor object, you can write to files on the tape. To open a stream to a magtape file call IOS_$OPEN or IOS_$CREATE, specifying the pathname of the magtape descriptor object. Note that when the IOS manager opens the file, it writes to the file specified by the file sequence number. By default, this file sequence number has the value of 1. When you are finished processing a tape file, you must close the stream, using IOS_$CLOSE. To access other files on the tape, you must first reopen (and close) the descriptor object to change the file sequence number. You change the sequence number by calling MTS_SET_ATTR specifying a new value for the MTS_$FILE_SEQUENCE_A attribute. To write to a tape file, call IOS_$PUT. (Specify an empty set of put options with [], since none of the options are meaningful for tape descriptor objects.)

The program in Example 9-3 accepts input from the user and writes it to two files on the tape. The program performs the following steps:

1. Declares a procedure to write to the tape files. This procedure creates a loop to get data from the user and write it to the tape file using IOS_$PUT until the user types CTRL/Z to terminate input. The program resets the stream marker's position after the user types CTRL/Z so that it no longer points to EOF, which prevents the program from returning an EOF error the next time it enters this procedure.

2. Creates a magtape descriptor object with the default attributes, using MTS_$CREATE_DEFAULT_DESC.

3. Sets the magtape NEWLINE-handling attribute (MTS_$ASCII_NL_A) to FALSE. This prevents the MTS manager from stripping NEWLINE characters as it writes each line to a tape file. By default, the MTS manager strips NEWLINE characters.

4. Closes the descriptor object, specifying the value of TRUE in the update parameter. This means that MTS_$CLOSE_DESC updates the file to reflect the changes made.

5. Opens a stream to the magtape file, using IOS_$OPEN.

6. Calls the procedure to write to a magtape file. It writes to the first file on the tape because, by default, the file sequence number has a value of 1.

7. Closes the stream to the magtape file.

8. Advances the file sequence number by:

   - Opening the descriptor object, using MTS_$OPEN_DESC with write access.

   - Getting the current file sequence number, using MTS_$GET_ATTR, and incrementing the number by 1.

   - Setting the number to the new value, using MTS_$SET_ATTR.

   - Closing the descriptor object, specifying TRUE to update it.

9. Repeats steps 5 - 7 to write to the second file on the tape.

```
PROGRAM ios_mts_write;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/mts.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

VAR
  {$CREATE_DEFAULT_DESC variables}
  status     : status_$t;
  pathname   : name_$pname_t;
  namelength : integer;
  handle     : mts_$handle_t;
  count      : integer;

  {$GET_ATTR variables}
  value      : mts_$attr_value_t;

  {STREAM_$OPEN variables}
  stream_id  : ios_$id_t;

  {$PUT_REC variables}
  buffer      : string;

{=== Procedure to check for errors. Prints error and exits on bad status.== }
PROCEDURE check_status;  { for error handling }
BEGIN
    IF (status.all <> status_$ok) THEN
    BEGIN
        error_$print( status );
        pgm_$exit;
    END;
END;

{=== Procedure to write to a tape file. ================================== }
PROCEDURE write_to_tape_file;
BEGIN

    { Get the input from the keyboard. }

    writeln;
    writeln ('Input data for the tape file:');
    writeln ('Or type CTRL/Z to quit.');
    readln (buffer);

    WHILE TRUE DO BEGIN
       { Write  to tape file with IOS_$PUT. }

       ios_$put ( stream_id,
                  [],
                  buffer,
                  SIZEOF(buffer),
                  status);
       check_status;
```

Example 9-3. Writing to a Magtape File

```
        writeln;
        writeln ('Input data for the tape file:');
        writeln ('Or type CTRL/Z to quit.');
        IF EOF THEN BEGIN
          { Reset the input pointer so that it won't point to EOF. }
          RESET (input);
          EXIT;
          END;
        readln (buffer);
        END;
END;
BEGIN {=== main ========================================================== }

    writeln;
    writeln ( 'This program gets input from you and writes it to a tape.');
    writeln;
    writeln ( 'The program first asks you to name a magtape descriptor ');
    writeln ( 'object that the program will create. ");
    writeln;
    writeln ( 'The program will then ask you to input data for the first ');
    writeln ( 'file and it will write that data to tape one line at a time. ');
    writeln ( 'It asks you to input data again, which it will write to the ');
    writeln ( 'second file. ');
    writeln;
    writeln ( 'To read the data from  tape, invoke the corresponding  ');
    writeln ( 'program,ios_mts_read.pas, specifying the magtape descriptor');
    writeln ( ' object that you created with this program.');

   { Create a magtape descriptor object with default values. }

   writeln ('Input a new descriptor tape file pathname:');
   namelength := SIZEOF(pathname);      { Max namelength }

    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);
    check_status;

    handle := mts_$create_default_desc ( pathname,
                                         namelength,
                                         status);
    check_status;

    { Turn off the NEWLINE handling. }
    value.b := FALSE;
    mts_$set_attr( handle,
                   mts_$ascii_nl_a,
                   value,
                   status);
    check_status;
```

Example 9-3. Writing to a Magtape File (Cont.)

```
mts_$close_desc (handle,
                 TRUE,    { Modify descriptor object }
                 status);
check_status;

{ Open the first tape file. }

stream_id := ios_$open ( pathname,
                         namelength,
                         [ios_$write_opt], { Write access }
                         status);
check_status;

{ Write to the tape file. }
write_to_tape_file;

{ Close the first tape file. }
ios_$close (stream_id,
            status);
check_status;

{ Change tape file number by opening the descriptor object for update. }

handle := mts_$open_desc (pathname,
                          namelength,
                          mts_$write, { Write access }
                          status);
check_status;

{ Get the current file number. }
mts_$get_attr (handle,
               mts_$file_sequence_a, { File sequence number }
               value,
               status);
check_status;

{ Increment the tape file sequence number. }
value.i := value.i + 1;

{ Set new file sequence number. }
mts_$set_attr (handle,
               mts_$file_sequence_a, { File sequence number }
               value,
               status);
check_status;

{ Close the descriptor object, modifying it to reflect the changes. }

mts_$close_desc (handle,
                 TRUE,
                 status);
check_status;
```

Example 9-3. Writing to a Magtape File (Cont.)

```
    { Open a second tape file. }

    stream_id := ios_$open ( pathname,
                             namelength,
                             [ios_$write_opt], { Write access }
                             status);

    check_status;

    { Write to the tape file. }
    write_to_tape_file;

    { Close second tape file. }

    ios_$close (stream_id,
                status);
    check_status;

END. { ios_mts_write }
```

Example 9-3. Writing to a Magtape File (Concluded)

9.4.5. Example of Reading from a Magtape File

To read from a tape file, use the IOS get calls, IOS_$GET or IOS_$LOCATE. Before you attempt to read from magtape ofile, you must set the file sequence number attribute to the number of the file that you want to read, using MTS_$SET_ATTR. The number of the first file on a tape is 1.

The program in Example 9-4 does the following:

1. Declares a procedure to read from files on a magtape using IOS_$GET.
2. Opens an existing magtape descriptor object specifying read access.
3. Sets the file sequence number to the first file on the tape which is number 1.
4. Closes the descriptor object, specifying a value of TRUE in the update parameter to update the changes.
5. Opens a stream to the tape file using IOS_$OPEN, specifying the pathname of the descriptor object.
6. Calls the procedure to read the tape. It asks the user for the number of lines to read from the magtape file.
7. Closes the stream to the magtape file, using IOS_$CLOSE.
8. Advances the file sequence number by reopening the tape descriptor object, changing the value of the MTS_$FILE_SEQUENCE_A attribute, and then closing the descriptor object.
9. Repeats steps 5 - 7 to read from the second file on the tape.

```
PROGRAM ios_mts_read (input,output);

%include '/ins/base.ins.pas';
%include '/sys/ins/ios.ins.pas';
%include '/sys/ins/mts.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';

CONST
  buffer_size = 256;
VAR
  {$OPEN_DESC variables}
  status      : status_$t;
  pathname    : name_$pname_t;
  namelength  : integer;
  handle      : mts_$handle_t;
  count       : integer;

  {$CLOSE_DESC variables}
  update      : boolean;

  {$OPEN variables}
  stream_id   : ios_$id_t;

  {$GET_ATTR variables}
  value       : mts_$attr_value_t;

  {GET variables}
  buffer       : string;
  ret_length   : integer32;

  get            : integer;
  number_of_recs : integer;

{ == Procedure to check for errors. Prints error and exits on bad status.== }
PROCEDURE check_status;
BEGIN
    IF (status.all <> status_$ok) THEN
    BEGIN
        error_$print( status );
        pgm_$exit;
    END;
END;
{ == Procedure to read tape files. ======================================== }
PROCEDURE read_from_tape_file;
BEGIN

   writeln ('Input the number of lines to read from the magtape file:');
   readln(number_of_recs);
```

Example 9-4. Reading from a Magtape File

```
   { Get records from tape file. }

   FOR get := 1 TO number_of_recs DO BEGIN

      ret_length := ios_$get (stream_id,
                              [],
                              buffer,
                              buffer_size,
                              status);
      check_status;

      { Write the record to standard output. }

      writeln (buffer : ret_length);
      writeln;

      END; {do}
END; { procedure }

BEGIN { == main ========================================================== }

    writeln;
    writeln ( 'This program reads data from two tape files.');
    writeln;
    writeln ( 'The program first asks you for to name the magtape descriptor ');
    writeln ( 'object of the tape you want to read data from. Specify the ');
    writeln ( 'name of the object that you created with the corresponding ');
    writeln ( 'program, ios_mts_write.pas. ');
    writeln;
    writeln ( 'The program will then ask you to specify the number of lines ');
    writeln ( 'you want to read data from the first file, and then writes ');
    writeln ( 'the data to the screen. It repeats the prompt for you to read ');
    writeln ( 'data from the second file. If you specify a number greater ');
    writeln ( 'than the number of lines in the file, the program terminates ');
    writeln ( ' with the end-of-file error status.');
    writeln;

    writeln ('Input the magtape descriptor object pathname:');
    namelength := SIZEOF(pathname);      { Max namelength }

    vfmt_$read2('%""%eka%.',
                count,
                status,
                pathname,
                namelength);
    check_status;

    { Set the file sequence number to the first file on the tape. }
    { Open the descriptor object for reset. }

    handle := mts_$open_desc (pathname,
                              namelength,
                              mts_$write, { Write access }
                              status);
    check_status;
```

Example 9-4. Reading from a Magtape File (Cont.)

```
{ Set file sequence number to 1. }
mts_$set_attr (handle,
               mts_$file_sequence_a, { File number}
               1,                    { Value}
               status);
check_status;

{ Close the descriptor object, keeping the changes. }
mts_$close_desc (handle,
                 TRUE,
                 status);
check_status;

{ Open the first file on the tape with read access. }

stream_id := ios_$open ( pathname,
                         namelength,
                         [],
                         status);
check_status;

{ Read from the tape file. }
read_from_tape_file;

{ Close the file. }

ios_$close (stream_id,
            status);
check_status;

{ Advance the tape file sequence number by opening }
{ the tape descriptor object for update. }
handle := mts_$open_desc (pathname,
                          namelength,
                          mts_$write, { Write access }
                          status);
check_status;

{ Get the current file sequence number. }
mts_$get_attr (handle,
               mts_$file_sequence_a, { File sequence number }
               value,
               status);
check_status;

{ Increment the tape file sequence number. }
value.i := value.i + 1;

{ Set new file number. }
mts_$set_attr (handle,
               mts_$file_sequence_a, { File number }
               value,
               status);
check_status;
```

Example 9-4. Reading from a Magtape File (Cont.)

```
    { Close the descriptor object, keeping the changes. }
    mts_$close_desc (handle,
                     TRUE,
                     status);
    check_status;

    { Open the second file on the tape with read access. }
    stream_id := ios_$open ( pathname,
                             namelength,
                             [],
                             status);
    check_status;

    { Read from the tape file. }
    read_from_tape_file;

    { Close the tape file. }
    ios_$close (stream_id,
                status);
    check_status;

END.{ ios_mts_read }
```

Example 9-4. Reading from a Magtape File (Concluded)

Appendix A
Sample Programs in C

Appendix A contains C translations of the Pascal examples that appear throughout this manual. You can also see these programs on-line. For details, see the Preface of this book.

Table A-1 summarizes the programs that appear in this appendix.

Table A-1. Summary of C Programs in Appendix A

Program Name	Description	Page
PFM_CLEAN_UP.C	Establishes a clean-up handler that deletes a file.	A-6
PGM_SHELL.C	Invokes the DATE Shell command using PGM_$INVOKE.	A-8
PGM_INVOKE.C	Invokes PGM_OPEN.BIN in wait mode.	A-9
PGM_OPEN.C	Invoked by PGM_INVOKE.C. It fails to open a file and passes a severity level indicating that an error occurred.	A-10
PGM_EC.C	Invokes two programs as child processes and uses eventcounts to wait for them.	A-11
PGM_INVOKE_DIVIDE.C	Invokes PGM_DIVIDE as a background process, passing an input and error stream.	A-14
PGM_DIVIDE.C	Divides two numbers and establishes a fault handler for the divide-by-zero fault. You must bind with the module, PGM_ZERO_HANDLER.	A-17
PGM_ZERO_HANDLER.C	Handles faults in a separate module for PGM_DIVIDE.	A-18
PGM_ORPHAN.C	Converts a child process into an orphan process.	A-19
PGM_PASS_ARGS.C	Invokes a child process and passes two arguments to it.	A-20
PGM_PASSEE_ARG.C	Accesses the second argument in a passed argument list using PGM_$GET_ARG. May be used with PGM_PASS_ARGS.	A-22

Table A-1. Summary of C Programs in Appendix A (Cont.)

Program Name	Description	Page
PGM_PASSEE.C	Accesses the arguments in a passed argument list using PGM_$GET_ARGS. May be used with PGM_PASS_STREAMS.	A-23
PGM_DEL_INV.C	Invoked by PASS_DEL.PAS. It retrieves arguments with GET_ARGS, deletes one with DEL_ARG, and invokes another program using the arg_vector.	A-24
PGM_PASS_STREAMS.C	Opens a file and passes its stream ID to PGM_PASSEE as standard output.	A-26
PGM_YOUR_PROC.C	Obtains process information about its own process.	A-28
PGM_CHILD_INFO.C	Invokes CALC.BIN as a child process and gets process information about it.	A-30
STREAM_INQ_REC_LEN.C	Inquires about the record length and type of a user-specified file. If the the file is fixed length, it opens the file.	A-32
STREAM_CHANGE_EXP.C	Opens a user-specified file, inquires about the explicit type attribute and sets it to TRUE if it is already TRUE.	A-34
STREAM_PUT_FIXED.C	Creates and writes a file of fixed-length employee records.	A-37
STREAM_PUT_VAR.C	Creates and writes a file of variable length employee records.	A-39
STREAM_PUT_VAR_UASC.C	Creates a UASC file and writes ASCII data to it.	A-41
STREAM_GET_VAR.C	Retrieves variable length records from the file created with STREAM_PUT_VAR.	A-43
STREAM_GET_VAR_UASC.C	Retrieves variable length records (lines) from the file created with STREAM_PUT_VAR_UASC.	A-45

Table A-1. Summary of C Programs in Appendix A (Cont.)

Program Name	Description	Page
STREAM_UPDATE.C	Permits you to update the address of the employee records in the file created with STREAM_PUT_FIXED. It writes each record with its record number. Then you indicate the record you want to update by record number.	A-48
STREAM_WRITE_TAPE.C	Creates a magtape descriptor file and writes two files to tape.	A-52
STREAM_READ_TAPE.C	Reads two files from tape.	A-56
STREAM_SIO_ACCESS.C	Opens an SIO line, inquires about the host-synch attribute, and changes it to FALSE, if necessary.	A-60
STREAM_MBX_CLIENT.C	Opens a stream to a mailbox, writes to it, and waits for a message -- all using the stream manager.	A-62
STREAM_LIST_LINKS.C	Reads a directory and lists the links in it, using the stream manager.	A-64
PAD_MAKE_WINDOWS.C	Creates and closes windows and window panes.	A-66
PAD_INQ_WINDOW_SIZE.C	Gets information about the size and position of windows open to the pad.	A-69
PAD_FULL_WINDOW_SHOW.C	Uses PAD calls to handle full windows. It returns the position of a window, including its border and legend.	A-71
PAD_WINDOW_SHOW.C	Uses PAD calls to pop and push windows, make a window visible and invisible, and remove a window border.	A-73
PAD_INQ_DISP_KBD.C	Gets the user's type of display and keyboard. It also redefines function keys.	A-77

Table A-1. Summary of C Programs in Appendix A (Cont.)

Program Name	Description	Page
PAD_SCALE.C	Creates two windows -- one by specifying line numbers, another by specifying raster units.	A-80
PAD_INQ_FONT.C	Creates a frame at the top of the user's standard output pad, and writes a prompt inside the frame.	A-83
PAD_DIGCLK.C	Displays a digital clock. You run this program with the DM CPO command. You can specify, in raster units, where you want the clock to be; otherwise, it appears in the upper left corner.	A-85
PAD_MAKE_ICON.C	Demonstrates PAD calls that use icons.	A-89
PAD_CREATE_ICON.C	Demonstrates PAD calls that create pads in icon format.	A-92
PAD_FILENAME.C	Creates a frame at the top of a window, and displays a filename in the inverted version of the current font.	A-96
PAD_RAW_MODE.C	Demonstrates using the input pad in raw mode. It asks for keyboard input, but does not echo the input to the screen.	A-102
PBUFS_PASTE_BUFFER.C	Asks the user to supply the name of a paste buffer. If it exists, the program displays the contents of the buffer. If it does NOT exist, the program creates a new paste buffer, and asks the user to write to it.	A-105

Table A-1. Summary of C Programs in Appendix A (Cont.)

Program Name	Description	Page
EC_TIME_KBD_EVENTS.C	Uses time and keyboard eventcounts.	A-108
EC_WAIT_FOR_TIME.C	Inhibits asynchrounous faults using a time eventcount.	A-111
CAL_DECODE_LOCAL.C	Gets the local date and time in readable format and writes it to the standard output.	A-114
TIME_ZONE.C	Gets a timezone offset by using the timezone name, then by using the time difference.	A-115
CAL_ADD_TIMES.C	Adds a specified number of seconds to the current local time.	A-117
CAL_SUB_TIMES.C	Subtracts an input absolute time from the current time.	A-118
TIME_COMPARE.C	Compares the modification times of two files to determine which was changed recently.	A-120
TIME_WAIT_ABS.C	Suspends execution until a specified time.	A-123
TIME_WAIT_OR_DEFAULT.C	Uses the time eventcount to cause a prompt to be output at 10-second intervals and "times out" after three prompts occur.	A-125

A.1. PFM_CLEAN_UP.C

```
/* PROGRAM pfm_cleanup */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/vfmt.ins.c"

status_$t           status;
boolean             stream_open;  /*state variable*/
short               count;  /*VFMT parameter*/

/* $CREATE variables */
name_$pname_t       pathname;
short               namelength;
stream_$id_t        stream_id;

/* $CLEANUP variable */
pfm_$cleanup_rec    handler_id;

/*******************************************************/
error_routine(status)  /* for error handling */

status_$t   status;
{
    error_$print(status);
    pgm_$exit();
}
/*******************************************************/
main()
{
/* initialize state variable */
    stream_open = false;    /* not open yet */

/* Clean-up handler code */
    status = pfm_$cleanup (handler_id); /* establish clean-up handler */

/* check for established status */
    if(status.all != pfm_$cleanup_set)
    {
/* delete file if open while fault occurs */
        if(stream_open)
            stream_$delete ( stream_id,
                             status);
        exit(1);;
    }
/* end of clean-up handler */

/* begin normal operations */

/* get the filename */
    printf("Input pathname: \n");
    gets(pathname);
    namelength = strlen(pathname);
```

```
    stream_$create ( pathname,
                     namelength,
                     stream_$write,              /* access      */
                     stream_$controlled_sharing, /* concurrency */
                     stream_id,
                     status);

    if(status.code != status_$ok)
        error_routine(status);

/* set state variable */
    stream_open = true;  /* file is open */

/*get the input */



/* finished processing the file */
/* release the clean-up handler  */

    pfm_$rls_cleanup (handler_id,
                      status);
}
```

A.2. PGM_SHELL.C

```
/* PROGRAM pgm_shell.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

pgm_$proc    handle;
status_$t    status;

/* declare and load the standarrd streams */
pgm_$connv   connv = { stream_$stdin, stream_$stdout,
                       stream_$errin, stream_$errout };

main()
{
    pgm_$invoke("/com/date",
                (short)9,
                (short)0, OL,  /* no args      */
                (short)4,
                connv,         /* std. streams */
                pgm_$wait,
                handle,
                status) ;

    check_status(status);

}/* end main */
/***************************************************************/
check_status(status)     /* for error_handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}
```

A.3. PGM_INVOKE.C

```
/* PROGRAM pgm_invoke.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

pgm_$proc    handle;
status_$t    status;

/* declare and load the standarrd streams */
pgm_$connv   connv = { stream_$stdin, stream_$stdout,
                       stream_$errin, stream_$errout };
/****************************************************************************/
main()
{
    pgm_$invoke("pgm_open.bin",
                (short)12,
                (short)0, 0L,  /* no args      */
                (short)4,
                connv,         /* std. streams */
                pgm_$wait,
                handle,
                status) ;

    check_status(status);
}/* end main */
/****************************************************************************/
check_status(status) /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {   switch(status.all)
        {   case pgm_$error :
                printf("Invoked program ended with an error status\n");
                break;
            case pgm_$warning :
                printf("Invoked program ended with an warning status\n");
                break;
        }/* end switch */
        pgm_$exit();
    }/* end if */
}/* end check_status() */
```

A.4. PGM_OPEN.C

```
/* PROGRAM open.c */

/* This program fails to open a non-existent file and */
/* returns an PGM_$ERROR severity                      */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"

status_$t         status;
name_$pname_t     pathname;
short             namelength;

/* $open variable */
stream_$id_t      stream_id;

main()
{
/* open the file */
    stream_$open ("file.out",
                  (short)9,
                  stream_$read,              /* access      */
                  stream_$controlled_sharing, /* concurrency */
                  stream_id,
                  status);

    check_status(status);
}

/**************************************************************/
check_status(status)    /* for error_handling */

status_$t   status;
{
    if(status.all != status_$ok)
    {
        pgm_$set_severity (pgm_$error);
        pgm_$exit();
    }
}/* end check_status */
```

A.5. PGM_EC.C

```
/* PROGRAM pgm_ec.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/ec2.ins.c"

#define calc1_ec     0
#define calc2_ec     1

ec2_$ptr_t          ec2_ptr[2];
long                ec2_val[2];
ec2_$eventcount_t   replace_ec;
short               which;
status_$t           status;
short               dead_count;
pgm_$proc           handle1;
pgm_$proc           handle2;

/* declare and load the standard streams */
pgm_$connv    connv = { stream_$stdin, stream_$stdout,
                        stream_$errin, stream_$errout };
/************************************************************************/
main()
{
/* invoke 1st process */
    pgm_$invoke("calc1.bin",  /* program name      */
                (short)9,     /* namelength        */
                (short)0, OL, /* no args           */
                (short)4,     /* number of streams */
                connv,        /* std. streams      */
                (short)0,     /* default mode      */
                handle1,      /* process handle    */
                status) ;

   check_status(status);

/* invoke 2nd process */
    pgm_$invoke("calc2.bin",
                (short)9,
                (short)0, OL,
                (short)4,
                connv,
                (short)0,
                handle2,
                status) ;
   check_status(status);

/* get ec for 1st process */
    pgm_$get_ec ( handle1,            /* process handle */
                  pgm_$child_proc,    /* ec key         */
                  ec2_ptr[calc1_ec],  /* ec_ptr         */
                  status);

    check_status(status);
```

```
/* get ec for 2nd process */
    pgm_$get_ec ( handle2,           /* process handle */
                  pgm_$child_proc,   /* ec key         */
                  ec2_ptr[calc2_ec], /* ec_ptr         */
                  status);

    check_status(status);

/* initialize the never ready event count */
    ec2_$init (replace_ec);

/* initialize counter */
    dead_count = 0;

/* initialize satisfaction values to 1 */
   ec2_val[calc1_ec] = 1;
   ec2_val[calc2_ec] = 1;

/* NOW GO INTO A LOOP PROMPTING FOR INPUT */

/* repeat until both processes complete */
    while(dead_count != 2)
    {   which = ec2_$wait ( ec2_ptr,    /* ec pointer array */
                            ec2_val,    /* ec value array   */
                            (short)2,   /* number of ec's   */
                            status);

        check_status(status);

/* decrement which to adjust for initial array index of 0 in C */
        which--;

        switch(which)
        {   case calc1_ec:  /* when process 1 completes... */
                printf("Processing Process 1 results\n");

/* get the termination status of calc1*/
                pgm_$proc_wait ( handle1,
                                 status);
                check_status(status);

/* load the pointer array with a valid pointer*/
                ec2_ptr[calc1_ec] = &replace_ec;

/* set the ec value to be 1 (process terminated)*/
                ec2_val[calc1_ec] = 1;
                break;

            case calc2_ec:
                printf("Processing Process 2 results\n");

/* get the termination status of calc2 */
                pgm_$proc_wait (handle2,
                                status);

                check_status(status);

/* load the pointer array with a valid pointer */
                ec2_ptr[calc2_ec] = &replace_ec;
```

```
/* set the ec value to be 1 (process terminated) */
                ec2_val[calc2_ec] = 1;
                break;
            }/* end switch */

/* advance the dead count */
        dead_count++;
    }/* end while */
}/* end main() */
/*************************************************************************/
check_status(status) /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}
```

A.6. PGM_INVOKE_DIVIDE.C

```
/* PROGRAM pgm_invoke_divide.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

status_$t    status;

/* $CREATE variables */

name_$pname_t   error_name;
short           error_len;
name_$pname_t   input_name;
short           input_len;
stream_$id_t    error_id;
stream_$id_t    input_id;
stream_$sk_t    seek_key;
char            number[20];
long            number_len;

/* PGM_$INVOKE variables */
pgm_$proc       handle;   /* process_handle    */
pgm_$connv      connv;    /* connection vector */
pinteger        arg_count;

main()
{

/* get standard error pathname for program to be invoked */
    printf("Input the filename to be opened as standard ");
    printf("error in background process DIVIDE:\n");
    gets(error_name);
    error_len = strlen(error_name);

/* create error file - get stream */

    stream_$create (error_name,
                    error_len,
                    stream_$write,       /* access    */
                    stream_$unregulated, /* conc      */
                    error_id,            /* stream-id */
                    status);

    check_status(status);

/* get standard input pathname for program to be invoked */
    printf("Input the filename to be opened as standard ");
    printf("input in background process DIVIDE:\n");
    gets(input_name);
    input_len = strlen(input_name);
```

```
/* create standard input file -- get stream */

    stream_$create (input_name,
                    input_len,
                    stream_$write,       /* access    */
                    stream_$unregulated, /* conc      */
                    input_id,            /* stream-id */
                    status);

    check_status(status);

/* Get numbers to be divided by invoked program and  */
/* write them to the created standard input file.   */
    printf("input an integer to be divided:\n");
    gets(number);
    number_len = strlen(number);

/* add one for the newline */
    number[++number_len] = '\n';    /* terminate w/ newline*/

/* write the number to the file */
    stream_$put_rec ( input_id,   /* stream to write to     */
                      &number,    /* address of data buffer */
                      number_len, /* length of data         */
                      seek_key,
                      status);

    check_status(status);

    printf("input an integer %.*s is to be divided by:\n",
          (number_len-1), number);
    gets(number);
    number_len = strlen(number);

/* add one for the newline */
    number[++number_len] = '\n';    /* terminate w/ newline*/

/*write the number to the file */
    stream_$put_rec ( input_id,
                     &number,
                      number_len,
                      seek_key,
                      status);

    check_status(status);

/* reset stream pointer to the beginning of the     */
/* input file before passing stream to the program */

    stream_$seek( input_id,          /* stream-id     */
                  stream_$rec,       /* seek-base     */
                  stream_$absolute,  /* seek-type     */
                  1L,                /* record number */
                  status);

    check_status(status);
```

```
/* load $INVOKE connection vector*/
    connv[0] = input_id;       /* set stream_id to be created stdin  */
    connv[1] = stream_$stdout; /* set stream_id to be STD_OUTPUT     */
    connv[2] = stream_$errin;  /* set stream_id to be STD_ERRIN      */
    connv[3] = error_id;       /* set stream_id to be created errout */

/* invoke program */
    pgm_$invoke ("pgm_divide",      /* pathname of program to invoke     */
                 (short)10,         /* length of pathname                */
                 (short)0,          /* number of arguments to be passed  */
                 0L,                /* no arguments                      */
                 (short)4,          /* number of streams to be passed    */
                 connv,             /* array of stream ids to be passed  */
                 pgm_$back_ground,  /* mode in which to invoke program   */
                 handle,            /* not used in background mode       */
                 status);           /* status                            */

    check_status(status);
}

/*****************************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}
```

A.7. PGM_DIVIDE.C

```
/* PROGRAM pgm_divide */

/* Program to divide two numbers */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/fault.ins.c"
#include "/sys/ins/error.ins.c"

short               number1;
short               number2;
status_$t           status;
pfm_$fh_handle_t    handler_id;

/* declare external fault handling function */
extern pfm_$fh_func_val_t zero_fault_handler();

main()
{
/* establish the zero divide handler     */

/* load the target fault - 1st parameter */
    handler_id = pfm_$establish_fault_handler ( fault_$zero_divide,
                                                (short)0,     /* default type */
                                                zero_fault_handler, /* This sends
                the ADDRESS of our fault handler! */
                                                status);
    if(status.all != status_$ok)
        error_$print(status);

/* read from standard input - (file passed by parent) */
    scanf("%hd",&number1);
    scanf("%hd",&number2);

/* calculate and write the result */
    printf("%d divided by %d is %d",number1, number2,(number1 / number2));
    printf(" with a remainder of %d\n", (number1 % number2));
}
```

A.8. PGM_ZERO_HANDLER.C

```
/* pgm_zero_handler.c */

/* This is a fault handling function that prints */
/* a line and continues to fault handle.         */

#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pfm.ins.c"

#define pfm_$Return_to_Peyton_Place pfm_$return_to_faulting_code


/* VERY IMPORTANT NOTE! The DOMAIN fault handler system code is
|| going to call the following routine when a fault occurs. Now,
|| that call will be done in "std_$call" format, because, of course,
|| that is the only kind of call it can make. But our fault
|| handler is REALLY a C function and therefore NOT created in
|| "std_$call" format. Since the DOMAIN code will push the address
|| of the 'pfm_$fault_rec_t', we, in C, must be prepared to accept
|| that pass-by-reference. So, notice that we declare OUR parameter
|| here as a pointer. */

pfm_$fh_func_val_t zero_fault_handler(f_status_p)

pfm_$fault_rec_t *f_status_p;
{
/* write a message to the error log */

    error_$print(f_status_p->status);
    return(pfm_$Return_to_Peyton_Place);

}/* end zero_fault_handler */
```

A.9. PGM_ORPHAN.C

```
/* PROGRAM pgm_orphan.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

uid_$t       puid;
status_$t    status;
pgm_$proc    handle;

/* declare and load the standard streams */
pgm_$connv   connv = { stream_$stdin, stream_$stdout,
                       stream_$errin, stream_$errout };

/*****************************************************************************/
main()
{
/* invoke child process */
    pgm_$invoke("pgm_shell.bin", /* program name   */
                (short)13,        /* namelength     */
                (short)0, 0L,    /* no args        */
                (short)4,
                connv,           /* std. streams   */
                (short)0,        /* default mode   */
                handle,          /* process handle */
                status) ;

   check_status(status);

/* communicate with child */

/* cut the child loose */

   pgm_$make_orphan( handle, /* process handle */
                     puid,   /* process uid    */
                     status);

   check_status(status);
}/* end main() */
/*****************************************************************************/
check_status(status) /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}
```

A.10. PGM_PASS_ARGS.C

```
/* PROGRAM pgm_pass_args */

/* This program passes a text string and the output stream */
/* to the program passee.pas                               */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

status_$t    status;

/* argument variables */
pgm_$arg     name, argument;

/* INVOKE variables */
pgm_$argv    argv;
pgm_$proc    handle;

/* declare and load the standarrd streams */
pgm_$connv   connv = { stream_$stdin, stream_$stdout,
                       stream_$errin, stream_$errout };

main()
{
/* load the arguments */
    strcpy(name.chars, "pgm_passee.bin");
    name.len =  strlen(name.chars);
    strcpy(argument.chars ,"test");
    argument.len = strlen(argument.chars);

/*load the argument vector w/ addresses*/
    argv[0] = (int *)&name;
    argv[1] = (int *)&argument;

    pgm_$invoke("pgm_passee.bin", /* process name          */
                (short)14,        /* name length           */
                (short)2,         /* arg count - name & arg */
                argv,             /* arg vector            */
                (short)4,         /* stream count          */
                connv,            /*std. streams           */
                (short)0,         /* mode                  */
                handle,           /* process handle        */
                status) ;

    check_status(status);

    pgm_$proc_wait ( handle,   /*process handle*/
                     status);

    check_status(status);
}/* end main */
```

```
/****************************************************************************/
check_status(status) /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {     error_$print(status);
          pgm_$exit();
    }
}
```

A.11. PGM_PASSEE_ARG.C

```
/* PROGRAM pgm_passee_arg.c */

/* This program is invoked by PASSER.PAS, it retrieves   */
/* argument w/ GET_ARG and writes to output              */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

status_$t   status;
pinteger    arg_length;     /* returned argument length       */
pinteger    arg_num;        /* ordinal # of desired argument  */
char        argument[256]; /* argument buffer                */
pinteger    max_len = 256; /* maximum length of returned arg */

main()
{
/* access 1st argument */
    arg_num = 0;  /* 1st arg */

    arg_length = pgm_$get_arg (arg_num,  /* arg number */
                               argument, /* arg buffer */
                               status,
                               max_len);

    check_status(status);

/* Null terminate the string */
    argument[arg_length] = '\0';
    printf("This is the first argument %s\n", argument);

/* access 2nd argument */
    arg_num = 1;   /* 2nd arg #, 0 is 1st */

    arg_length = pgm_$get_arg (arg_num,  /* arg number */
                               argument, /* arg buffer */
                               status,
                               max_len);

    check_status(status);

/* Null terminate the string */
    argument[arg_length] = '\0';
    printf("This is the second argument %s\n", argument);
}/* end main() */
/***************************************************************************/
check_status(status) /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         exit(1);
    }
}
```

A.12. PGM_PASSEE.C

```
/* PROGRAM pgm_passee.c */

/* This program is invoked by PASSER.PAS, it retrieves    */
/* arguments w/ GET_ARGS and writes to output             */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

/* declare an explicit argument pointer */
typedef
    pgm_$arg    *pgm_arg_ptr;


pinteger        arg_count;    /* argument count  */
pgm_$argv_ptr   arg_vec_addr; /* argument vector */
pinteger        i;            /* index           */

/* declare array to hold arguments */
pgm_arg_ptr     arguments[128];


main()
{
/* get a pointer to the argument array */

    pgm_$get_args (arg_count,     /* number of arguments */
                   arg_vec_addr); /* returned pointer    */

    for(i=0; i<arg_count; i++)
    {
/* typecast the pointer and load into argument array */
        arguments[i] = (pgm_arg_ptr) (*arg_vec_addr)[i];

/* write argument to output (dereference explicit pointer) */

/* null terminate string */
        arguments[i]->chars[arguments[i]->len] = '\0';
        printf("Argument %d is  %s\n", i, arguments[i]->chars);
    }/* end for */
}/* end main */
```

A.13. PGM_DEL_INV.C

```
/* PROGRAM pgm_del_inv.c */

/* This program is invoked by PASS_DEL.PAS, it retrieves    */
/* arguments w/ GET_ARGS deletes one w/ DEL_ARG and invokes */
/* another program using the arg_vector                     */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

/* construct a pointer to arguments */
typedef
    pgm_$arg    *pgm_arg_ptr;

pinteger    arg_count;
pgm_$argv_ptr    arg_vec_addr;

/* declare array to hold arguments */
pgm_arg_ptr arguments[128];

short    i;
/* INVOKE variables */
status_$t    status;
pgm_$proc    handle;

/* declare and load the standrrd streams */
pgm_$connv   connv = { stream_$stdin, stream_$stdout,
                       stream_$errin, stream_$errout };

main()
{
    printf("In del_inv\n");

    pgm_$get_args ( arg_count,     /* number of arguments        */
                    arg_vec_addr); /* pointer to argument vector */


    printf("passed folowing arguments:\n");

    for(i=0; i<arg_count; i++)
    {
        arguments[i] = (pgm_arg_ptr) (*arg_vec_addr)[i];
        arguments[i]->chars[arguments[i]->len] = '\0';
        printf("ARG %d  %s\n", i, arguments[i]->chars);
    }
    printf("\n");
    printf("deleting arg 0\n");

/* delete program name argument */
    pgm_$del_arg(0);

/* get_args passes integer pointers. To reference you must */
/* type cast the integer pointers to pgm_$arg pointers.    */
    for(i=0; i<arg_count; i++)
        arguments[i] = (pgm_arg_ptr) (*arg_vec_addr)[i];
```

```
/* returned strings must be null terminated */
        arguments[0]->chars[arguments[0]->len] = '\0';
        printf("invoking %s (now arg 0)\n", arguments[0]->chars);

/* invoke second program w/ modified arg vector */

    pgm_$invoke( arguments[0]->chars, /* process name      */
                 arguments[0]->len,   /* name length       */
                 (short)1,            /* arg count - name */
                 *arg_vec_addr,       /* arg vector        */
                 (short)4,            /* stream count      */
                 connv,               /* std streams       */
                 pgm_$wait,           /* mode              */
                 handle,              /* process handle    */
                 status);

  check_status(status);
}/* end main() */
/***************************************************************************/
check_status(status)   /* for error_handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}
```

A.14. PGM_PASS_STREAMS.C

```
/* PROGRAM pgm_pass_streams.c */

/* This program passes a stream to an invoked program */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"


status_$t    status;

/* argument variables */
pgm_$arg        name, argument;

/* INVOKE variables */
pgm_$argv       argv;
pgm_$connv      connv;
pgm_$proc       handle;

/* CREATE variables */
name_$pname_t   pathname;
short           namelength;
stream_$id_t    stream_id;


main()
{
/* get the input  */
    printf("Enter the output file pathname:\n");
    gets(pathname);
    namelength = strlen(pathname);

/* open w/ $CREATE */

    stream_$create( pathname,
                    namelength,
                    stream_$write,              /* access */
                    stream_$controlled_sharing, /* conc   */
                    stream_id,
                    status);

    check_status(status);

/* load the arguments */
    strcpy(name.chars, "pgm_passee.bin");
    name.len = strlen(name.chars);
    strcpy(argument.chars ,"test");
    argument.len = strlen(argument.chars);

/* load the argument vector w/ addresses */
    argv[0] = (int *)&name;
    argv[1] = (int *)&argument;
```

```
/* load connection vector */
    connv[0] = stream_$no_stream; /* null stream             */
    connv[1] = stream_id;         /* pass stream-id as std out */
    connv[2] = stream_$errin;
    connv[3] = stream_$errout;

    pgm_$invoke("pgm_passee.bin", /* process name          */
                (short)14,        /* name length           */
                (short)2,         /* arg count - name & arg */
                argv,             /* arg vector            */
                (short)4,         /* stream count          */
                connv,            /* connection vector     */
                (short)0,         /* mode                  */
                handle,           /* process handle        */
                status) ;

    check_status(status);

/* get process termination status */
    pgm_$proc_wait( handle,       /* process handle */
                    status);

    check_status(status);

}/* end main() */
/***************************************************************************/
check_status(status)    /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}
```

A.15. PGM_YOUR_PROC.C

```
/* PROGRAM pgm_your_proc.c */

/* This program gets the home directory, SID and UID of     */
/* the calling process and gets the information record of   */
/* calling process. It also gets the total CPU time it uses.*/

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/proc1.ins.c"
#include "/sys/ins/proc2.ins.c"
#include "/sys/ins/pm.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/type_uids.ins.c"

char                    home[80];
pinteger                home_len;
char                    sid[80];
pinteger                sid_len;
uid_$t                  uid;
proc2_$info_t           info;
status_$t               status;
time_$clock_t           total_time;
cal_$timedate_rec_t     d_clock;

main()
{   pm_$get_home_txt ((short)30,  /* maxlen */
                      home,       /* dir    */
                      home_len);

/* null terminate returned strings before printing */
    home[home_len] = '\0';
    printf("home directory %s\n", home);

    pm_$get_sid_txt ((short)40,  /* maxlen */
                     sid,        /* dir    */
                     sid_len);

/* null terminate returned strings before printing */
    sid[sid_len] = '\0';
    printf("sid %s\n", sid);

    proc2_$who_am_i (uid);

    printf("uid %d  %d\n", uid.high, uid.low);

    proc2_$get_info (uid,       /* process uid        */
                     info,
                     (short)36, /* info buffer length */
                     status);

    if(status.all != proc2_$is_current)
        error_$print(status);
```

```
/* write the information */
    printf("\nstack uid %d\n", info.stack_uid.high);
    printf("stack uid %d\n", info.stack_uid.low);
    printf("stack base %d\n", info.stack_base);

    if(proc2_$waiting & info.state)
        printf("state : waiting\n");
    if(proc2_$suspended & info.state)
        printf("state : suspended\n");
    if(proc2_$susp_pending & info.state)
        printf("state : susp_pending\n");
    if(proc2_$bound & info.state)
        printf("state : bound\n");

    printf("user sr %d\n", info.usr);
    printf("user pc %d\n", info.upc);
    printf("user stack pointer %d\n", info.usp);
    printf("sb ptr %d\n", info.usb);

/*decode the time*/
    cal_$decode_time (info.cpu_total,
                      d_clock);

    printf ("cum cpu: %d %d %d\n", d_clock.hour,
                                   d_clock.minute,
                                   d_clock.second);
    printf("priority %d\n\n", info.priority);

    proc1_$get_cput (total_time);

/*decode the time*/
    cal_$decode_time (total_time,
                      d_clock);

    printf ("GET_CPU total time: %d %d %d\n", d_clock.hour,
                                   d_clock.minute,
                                   d_clock.second);

}/* end main */
```

A.16. PGM_CHILD_INFO.C

```
/* PROGRAM pgm_child_info.pas */

/* This program gets the amount of time the child has used */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/proc2.ins.c"
#include "/sys/ins/error.ins.c"

status_$t            status;
uid_$t               proc_uid;   /* process uid            */
proc2_$info_t        info;       /* information record     */
time_$clock_t        total_time; /* encoded time           */
cal_$timedate_rec_t  d_clock;    /* decoded time           */
time_$clock_t        rel_time;   /* relative amount of time */
pgm_$proc            handle;     /* process handle         */

/* declare and load the standard streams */
pgm_$connv    connv = { stream_$stdin, stream_$stdout,
                        stream_$errin, stream_$errout };
/***************************************************************************/
main()
{
    pgm_$invoke("calc.bin",  /* process name   */
                (short)8,    /* name length    */
                (short)0,0L, /* no args        */
                (short)4,    /* stream count   */
                connv,       /* std streams    */
                (short)0,    /* default mode   */
                handle,      /* process handle */
                status) ;

    check_status(status);

/* wait 10 seconds */

/* convert # of seconds to UTC value */
    cal_$sec_to_clock ( 10L,
                        rel_time);

    time_$wait ( time_$relative,     /* pre-defined  */
                 rel_time,           /* time to wait */
                 status);

    check_status(status);

/* resume */
```

```
/* get the process uid */

    pgm_$get_puid ( handle,    /* process handle */
                    proc_uid,  /* process uid    */
                    status);

    check_status(status);

/* get process information */

    proc2_$get_info ( proc_uid,   /* process uid       */
                      info,
                      (short)36,  /*info buffer length */
                      status);

    check_status(status);

/* decode the cpu time */

    cal_$decode_time ( info.cpu_total,
                       d_clock);

    printf("Accumulated CPU time of Child %d %d %d\n", d_clock.hour,
                                                        d_clock.minute,
                                                        d_clock.second);

/* get child's terminaton status */

    pgm_$proc_wait ( handle,        /* process handle */
                     status);

    check_status(status);
}/* end main() */
/****************************************************************************/
check_status(status)  /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         pgm_$exit();
    }
}/* end check_status */
```

A.17. STREAM_INQ_REC_LEN.C

```
/* PROGRAM stream_inq_rec_len.c */

/* This program inquires about the attributes of a file.   */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

#define SETSIZE (short)32

status_$t                status;
name_$pname_t            pathname;
short                    namelength;
short                    rec_len;

/* INQUIRE variables */
stream_$inquire_mask_t   input_mask;
stream_$ir_opt           inquiry_type;
stream_$ir_rec_t         attributes;
stream_$inquire_mask_t   error_mask;

/* $OPEN variables */
stream_$opos_t           access;
stream_$omode_t          conc;
stream_$id_t             stream_id;

error_routine()     /* for error handling */
{
    pgm_$set_severity(pgm_$error);
    pgm_$exit();
}
/***********************************************/
test_error_mask(error_mask)

stream_$inquire_mask_t   error_mask;
{
    if(lib_$member_of_set(error_mask, SETSIZE, stream_$rec_lgth))
    {
        printf("stream_$rec_lgth in error mask\n");
        error_routine();
    }
    if(lib_$member_of_set(error_mask, SETSIZE, stream_$rec_type))
    {
        printf("stream_$rec_type in error mask\n");
        error_routine();
    }
}/*test_error_mask*/
/***********************************************/
main()
{
/* get the filename */
    printf("Input the pathname\n");
    gets(pathname);
    namelength  = strlen(pathname);
```

```
/*load name and length into attribute record*/
    strcpy(attributes.obj_name , pathname);
    attributes.obj_namlen = namelength;

/*get file info by pathname, even though not open*/
    inquiry_type = stream_$name_unconditional;

/*set attribute bits in mask*/
    lib_$init_set(input_mask, SETSIZE);

/* length of largest or fixed record      */
    lib_$add_to_set(input_mask, SETSIZE, (short)stream_$rec_lgth);

/* record type, fixed, variable, or undef */
    lib_$add_to_set(input_mask, SETSIZE, (short)stream_$rec_type);

/*inquire by name*/

    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);

    if(status.all != status_$ok)
        error_routine();

/* check the error mask */
    test_error_mask(error_mask);

/* test the record type for fixed length */
    if((stream_$rtype_t)attributes.rec_type == stream_$f2)
        rec_len = attributes.rec_lgth;
    else
        error_routine();

/*open the file*/
    access = stream_$read;
    conc = stream_$controlled_sharing;

    stream_$open( pathname,
                  namelength,
                  access,
                  conc,
                  stream_id,
                  status);

    if(status.all != status_$ok)
        error_routine();

    stream_$close( stream_id,
                   status);

    if(status.all != status_$ok)
        error_routine();
}
```

A.18. STREAM_CHANGE_EXP.C

```
/* PROGRAM stream_change_exp.c */

/* This program inquires about the attributes of a file, */
/* redefines an attribute and and inquires to check that */
/* the change took place.                                */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

#define SETSIZE (short)32

status_$t    status;
name_$pname_t    pathname;
short    namelength;

/* INQUIRE variables */
stream_$inquire_mask_t  input_mask;
stream_$ir_opt          inquiry_type;
stream_$ir_rec_t        attributes;
stream_$inquire_mask_t  error_mask;

/* $OPEN variables */
stream_$opos_t          access;
stream_$omode_t         conc;
stream_$id_t            stream_id;

/* REDEFINE variables */
stream_$redef_mask_t    redef_mask;
stream_$redef_mask_t    redef_error_mask;
/**********************************************************/
main()
{
/* get the filename */
    printf("Input the pathname\n");
    gets(pathname);
    namelength = strlen(pathname);

/* open the file */
    access = stream_$write;
    conc = stream_$controlled_sharing;

    stream_$open (pathname,
                  namelength,
                  access,
                  conc,
                  stream_id,
                  status);

    if(status.all != status_$ok)
        error_routine();
```

```
/* set proper INQUIRE arguments */

/* set info bits in mask */
/*set attribute bits in mask*/
    lib_$init_set(input_mask, SETSIZE);

/*explicit type bit*/
    lib_$add_to_set(input_mask, SETSIZE, (short)stream_$explicit_type);

    inquiry_type = stream_$use_strid; /* get info by stream    */
    attributes.strid = stream_id;    /* provide the stream id */

/* inquire by stream */

    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);

    if((status.all != status_$ok) ||
       (lib_$member_of_set(error_mask, SETSIZE, stream_$explicit_type )))
        error_routine();

    printf
     ("Explicit type was %s\n",( attributes.explicit_type) ? "TRUE" : "FALSE");

/*test returned explicit type*/
    if(! attributes.explicit_type)
    {
/*set redefinition mask*/
    lib_$init_set(redef_mask, SETSIZE);
    lib_$add_to_set(redef_mask, SETSIZE, (short)stream_$explicit_type);

/* redefine explicit type */
        attributes.explicit_type = true;

/* change the type */

        stream_$redefine( stream_id,
                          redef_mask,
                          attributes,
                          redef_error_mask,
                          status);

    if((status.all != status_$ok) ||
       (lib_$member_of_set(redef_error_mask, SETSIZE, stream_$explicit_type )))
            error_routine();
    }/*if*/
```

```
/* inquire by stream */

    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);

    if((status.all != status_$ok) ||
       (lib_$member_of_set(error_mask, SETSIZE, stream_$explicit_type )))
        error_routine();

    printf
    ("Explicit type is now %s\n",(attributes.explicit_type) ? "TRUE" : "FALSE");
}
/*******************************************************/
error_routine()
{
    pgm_$set_severity(pgm_$error);
    pgm_$exit();
}/* end error_routine */
```

A.19. STREAM_PUT_FIXED.C

```
/* PROGRAM stream_put_fixed.c */

/* This program creates a file using stream_$create_bin and    */
/* prompts for input to be written as fixed length records.    */
/* The address field can be modified using the program update. */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

typedef struct
{   short    emp_id;
    char     address[80];
    char     name[80];
}info_rec_t;

status_$t        status;
name_$pname_t    pathname;
short            namelength;
info_rec_t       info_rec; /*data_buffer*/
char             c;

/* create variables */
stream_$opos_t   access;
stream_$omode_t  conc;
stream_$id_t     stream_id;

/* put variables */
stream_$sk_t     seek_key;
long             buflen;
/*******************************************************/
main()
{
/* get the filename */
    printf("Input the pathname file to be created:\n");
    gets(pathname);

    namelength = strlen(pathname);

/* create the file */
    access = stream_$write;
    conc = stream_$controlled_sharing;

    stream_$create_bin (pathname,
                        namelength,
                        access,
                        conc,
                        stream_id,
                        status);
    check_status(status);
```

```
/* get record info */
    printf("Input employee name (or ctl z to stop):\n");
    while(gets(info_rec.name) != NULL)
    {
        printf("Input employee id #:\n");
        scanf("%hd", &info_rec.emp_id);
        for(c = getchar(); c != '\n'; c = getchar());
        printf("Input address of employee:\n");
        gets(info_rec.address);

/* write record */
        buflen = sizeof(info_rec);  /* Record length is fixed */

        stream_$put_rec ( stream_id,  /* stream-id of open file */
                          &info_rec,  /* pointer to data buffer */
                          buflen,     /* length of data buffer  */
                          seek_key,   /* returned seek key      */
                          status);

        check_status(status);
        printf("Record written\n");
        printf("Input next employee name (or ctl z to stop):\n");
    }/*while*/
}
/*****************************************************/
check_status(status)  /* for error handling */

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status */
```

A.20. STREAM_PUT_VAR.C

```
/* PROGRAM stream_put_var */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"

/* define employee record */
typedef struct
{   short    namelen;
    short    emp_id;
    char    name[80];
}info_rec_t;

status_$t        status;
name_$pname_t    pathname;
short            namelength;
info_rec_t       info_rec;
char             c;

/* $OPEN variables */
stream_$opos_t   access;
stream_$omode_t  conc;
stream_$id_t     stream_id;

/* put variables */
stream_$sk_t     seek_key;
long             buflen;
/************************************************************/
main()
{
/* get the filename */
    printf("Input the pathname file to be created:\n");
    gets(pathname);

    namelength = strlen(pathname);

/* create a record-oriented  file */
    access = stream_$write;
    conc = stream_$controlled_sharing;

    stream_$create_bin( pathname,
                        namelength,
                        access,
                        conc,
                        stream_id,
                        status);
    check_status(status);

/* get record info */
    printf("Input employee name (or ctl z to stop):\n");
    while((gets(info_rec.name)) != NULL)
    {
        put_name_length();
        printf("Input employee id #:\n");
        scanf("%hd", &info_rec.emp_id);
```

```
/* eat the newline */
        for(c = getchar(); c != '\n'; c = getchar());

/* put employee id field in the record */
        buflen = sizeof(info_rec.emp_id);

        stream_$put_chr ( stream_id,       /* id of open stream */
                         &info_rec.emp_id, /* pointer to buffer */
                          buflen,          /* length of buffer  */
                          seek_key,        /* returned key      */
                          status);

/* write name field and terminate record */
        buflen = info_rec.namelen;   /* Record length varies with*/
                                     /* length of name field     */

        stream_$put_rec ( stream_id,
                         &info_rec.name,
                          buflen,
                          seek_key,
                          status);

        check_status(status);
        printf("record written\n");
        printf("Input next employee name (or ctl z to stop):\n");
    }/* end while */
}
/*********************************************************/
/* Procedure to calculate the length of the  name and put  */
/* the namelen field into the record using STREAM_$PUT_CHR */
/*********************************************************/
put_name_length()
{
/* calculate the length of info_rec.name */
    info_rec.namelen = strlen(info_rec.name);

    buflen = sizeof(info_rec.namelen);

    /* put the namelength in the record */

    stream_$put_chr ( stream_id,        /*id of open file*/
                     &info_rec.namelen, /*pointer to buffer*/
                      buflen,           /*length of buffer*/
                      seek_key,         /*returned key*/
                      status);

    check_status(status);
}/*put_name_length*/
/*********************************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}
```

A.21. STREAM_PUT_VAR_UASC.C

```
/* PROGRAM stream_put_var_uasc.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"


status_$t        status;
name_$pname_t    pathname;
short            namelength;
char             line[80];

/* $OPEN variables */
stream_$opos_t   access;
stream_$omode_t  conc;
stream_$id_t     stream_id;

/* put variables */
stream_$sk_t     seek_key;
long             buflen;

main()
{
/* get the filename */
    printf("Input the pathname file to be created:\n");
    gets(pathname);

/* calculate the length of pathname */
    namelength = strlen(pathname);

/*create the file*/
    access = stream_$write;
    conc = stream_$controlled_sharing;

    stream_$create (pathname,
                    namelength,
                    access,
                    conc,
                    stream_id,
                    status);

    check_status(status);

/* get a line of input */
    printf("Input data (or ctl z to stop):\n");
    while(gets(line) != NULL)
    {
        buflen = strlen(line);
/*terminate line with newline character*/
        line[buflen] = '\n';
        buflen++;
```

```
/*write the line to a file*/

        stream_$put_rec ( stream_id,
                         &line,
                          buflen,
                          seek_key,
                          status);

        check_status(status);
        printf("Record written\n");
        printf("Input more info (or ctl z to stop):\n");
    }/* end while */
}/* end main */

check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}
```

A.22. STREAM_GET_VAR.C

```
/* PROGRAM stream_get_var */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

/* define record buffer */
typedef struct
{   short    namelen;
    short    emp_id;
    char    name[80];
}info_rec_t;

status_$t      status;
name_$pname_t  pathname;
short          namelength;
info_rec_t     info_rec;
info_rec_t    *retptr;
long           retlen;

/* $OPEN variables */
stream_$id_t   stream_id;

/* $GET variables */
stream_$sk_t   seek_key;
long           buflen;

main()
{
/* get the filename */
    printf("Input the pathname of record structured file to be read:\n");
    gets(pathname);
    namelength = strlen(pathname);

/* open the file for reading */

    stream_$open (pathname,
                  namelength,
                  stream_$read,               /* access */
                  stream_$controlled_sharing, /* conc   */
                  stream_id,
                  status);

    if(status.all != status_$ok)
       error_routine();
```

```
/* Enter loop to get and print  records */
    while(status.all == status_$ok)
    {
/* get a record */

        stream_$get_rec( stream_id,
                        &info_rec,
                         (long)sizeof(info_rec),
                         retptr,
                         retlen,
                         seek_key,
                         status);

/* test for EOF */
        if((status.code == stream_$end_of_file) &&
           (status.subsys == stream_$subs))
            break;
        if(status.all != status_$ok)
            error_routine();

/* assign returned pointer to buffer */
        info_rec = *retptr;

/* print the name and id fields */
        printf("\nname: %.*s\n", info_rec.namelen, info_rec.name);
        printf("id: %d\n", info_rec.emp_id);
    }/* end while */
}

error_routine()
{
    pgm_$set_severity(pgm_$error);
    pgm_$exit();
}
```

A.23. STREAM_GET_VAR_UASC.C

```
/* PROGRAM stream_get_var_uasc.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

#define max_lines     1024 /* maximum number of lines */

typedef
    char  string[80];

status_$t         status;
name_$pname_t     pathname;
short             namelength;
string            line;
string           *retptr;
long              retlen;
char              c;

/* $OPEN variables */
stream_$id_t      stream_id;

/* $GET variables */
long              buflen;
short             choice_line;
short             no_of_lines;

/* $SEEK variables */
stream_$sk_t      seek_key;
stream_$parm1_t   seek_base;
stream_$parm2_t   seek_type;

/* declare vector to hold seek keys */
stream_$sk_t  seek_vector[max_lines];
/**************************************************/
main()
{
/* get the filename */
    printf("Input the pathname of uasc file to be read:\n");
    gets(pathname);
    namelength = strlen(pathname);

/* open the file for reading */

    stream_$open (pathname,
                  namelength,
                  stream_$read,                /* access */
                  stream_$controlled_sharing,  /* conc   */
                  stream_id,
                  status);

    if(status.all != status_$ok)
       error_routine();
```

```
/* read the file and fill the seek_vector with seek keys */

    no_of_lines = 0;
    while(status.all == status_$ok) /* while there is input */
    {
/* read a line */
        stream_$get_rec ( stream_id,
                          &line,
                           (long)sizeof(line),
                           retptr,
                           retlen,
                           seek_key,
                           status);

/* test for EOF */
        if((status.code == stream_$end_of_file) &&
           (status.subsys = stream_$subs))
            break;

        if(status.all != status_$ok)
            error_routine();

/* increment the vector index */
        no_of_lines++;

/* test for maximum nuber of lines */
        if(no_of_lines <= max_lines)

/*load vector with the returned seek key*/
            seek_vector[no_of_lines - 1] = seek_key;
        else
        {
            printf("maximum number of lines exceeded\n");
            break;
        }

    }/* end while*/

/* prompt the user for a line number */
    printf( "What number line would you like to see");
    printf(" (1 - %d) (ctl z to stop):\n", no_of_lines);

    while((scanf("%hd", &choice_line)) != EOF)
    {

/* eat the newline */
        for(c=getchar(); c != '\n'; c = getchar());
/* test to see if the chosen line is in range */
        while((choice_line <= 0) || (choice_line > no_of_lines))
        {
            printf("line number out of range enter a number");
            printf(" between 1 and %d:\n", no_of_lines);
            scanf("%hd", &choice_line);

/* eat the newline */
            for(c = getchar(); c != '\n'; c = getchar());
        }/* end while */
```

```
/*load the seek key using the vector*/
        seek_key = seek_vector[choice_line - 1];

        stream_$seek ( stream_id,
                       stream_$key,      /* seek_base */
                       stream_$absolute, /* seek_type */
                       seek_key,
                       status);

        if(status.all != status_$ok)
            error_routine();

/* read the line */
        stream_$get_rec ( stream_id,
                          &line,
                          (long)sizeof(line),
                          retptr,
                          retlen,
                          seek_key,
                          status);

        if(status.all != status_$ok)
            error_routine();

/* print the line */
        printf("%.*s\n",retlen, retptr);

/* prompt for next line */
        printf( "What number line would you like to see");
        printf(" (1 - %d) (ctl z to stop):\n", no_of_lines);

    }/* end while */
}/* end main */
/**************************************************/
error_routine()         /* for error handling */
{
    pgm_$set_severity(pgm_$error);
    pgm_$exit();
}
```

A.24. STREAM_UPDATE.C

```
/* PROGRAM stream_update.c */

/* This program opens a stream to a file of fixed length records   */
/* created by the program put_fixed.  The records are sequentially */
/* counted and written. The user is given the option of randomly   */
/* modifying the address of any employee.                          */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"

#define SETSIZE (short)32

/* define the record buffer */
typedef struct
{   short    emp_id;
    char     address[80];
    char     name[80];
}info_rec_t;

status_$t                status;
name_$pname_t            pathname;
short                    namelength;
info_rec_t               info_rec;
info_rec_t              *retptr;
long                     retlen;
stream_$id_t             stream_id;

/* $GET  variables */
stream_$sk_t             seek_key;
long                     buflen;

/* INQUIRE/REDEFINE variables*/
stream_$inquire_mask_t   input_mask;
stream_$ir_opt           inquiry_type;
stream_$ir_rec_t         attributes;
stream_$inquire_mask_t   error_mask;
long                     choice_rec; /* the record number user wants changed */
short                    no_of_recs; /* # of records in file                 */
char                     response;   /* Y to modify record, N to leave as is */
char                     c;

main()
{
/* get the filename */
    printf("Input pathname of record structured file to be updated:\n");
    gets(pathname);
    printf("\n");
    namelength = strlen(pathname);
```

```
/* open the file */

    stream_$open ( pathname,
                   namelength,
                   stream_$update,       /* update access    */
                   stream_$no_conc_write, /* no other writers */
                   stream_id,
                   status);
    check_status(status);

/* set explicit move mode */

    lib_$init_set(input_mask, SETSIZE);

    lib_$add_to_set(input_mask, SETSIZE, (short)stream_$explicit_ML);

    attributes.explicit_ml = true;

    stream_$redefine ( stream_id,
                       input_mask,
                       attributes,
                       error_mask,
                       status);
    check_status(status);

    no_of_recs = 0;

/* Read and print records and record numbers*/
/* while there is input and no problems     */
    while(status.all == status_$ok)
    {   buflen = sizeof(info_rec);  /* Record length is fixed */

        stream_$get_rec ( stream_id,
                         &info_rec,
                          (long)sizeof(info_rec),
                          retptr,
                          retlen,
                          seek_key,
                          status);
        if((status.code == stream_$end_of_file) &&
           (status.subsys == stream_$subs))
            break;
        else
            check_status(status);

 /* increment and print the record number */
        no_of_recs++;
        printf("Record # %d\n", no_of_recs);

/* load the record buffer */
        info_rec = *retptr;

/* print the employee id, name and address */
        printf("employee # %d\n" ,info_rec.emp_id);
        printf("name: %s\n", info_rec.name);
        printf("address: %s\n\n", info_rec.address);

    }/* end while*/
```

```
/* update the addresses */

    printf("What number record would you like to update");
    printf(" (1 - %d) (ctl z to stop):\n", no_of_recs);

    while(scanf("%d", &choice_rec) != EOF)
    {

/* eat the newline */
        for(c = getchar(); c != '\n'; c = getchar());

/* test record choice */
        while((choice_rec <= 0) || (choice_rec > no_of_recs))
        {
            printf("record number out of range enter a number");
            printf(" between 1 and %d :\n", no_of_recs);
            scanf("%d", &choice_rec);

/* eat the newline */
            for(c = getchar(); c != '\n'; c = getchar());
        }/* end while */

 /* position to specified record - absolute record seek */

        stream_$seek ( stream_id,
                       stream_$rec,       /*seek_base*/
                       stream_$absolute,  /*seek_type*/
                       choice_rec,        /*offset   */
                       status);
        check_status(status);

/* read the record */
        stream_$get_rec ( stream_id,
                          &info_rec,
                          (long)sizeof(info_rec),
                          retptr,
                          retlen,
                          seek_key,
                          status);

        check_status(status);

/* load the record buffer */
        info_rec = *retptr;

/* print the employee id, name and address */
        printf("employee # %d\n",info_rec.emp_id);
        printf("name: %s\n", info_rec.name);
        printf("address: %s\n\n", info_rec.address);

/* prompt for confirmation */
        printf("Would you like to update the address?");
        printf(" (Y or N): \n");
        response = getchar();
```

```
/* eat the newline */
        for(c = getchar(); c != '\n'; c = getchar());

        if((response == 'Y') || (response == 'y'))
        {
            printf("Enter the new address (on one line): ");
            gets(info_rec.address);

/* reposition to beginning of the record */

            stream_$seek ( stream_id,
                           stream_$rec,      /* seek_base */
                           stream_$absolute,/* seek_type */
                           choice_rec,      /* offset    */
                           status);

            check_status(status);

/* update the record */

            stream_$replace ( stream_id,
                             &info_rec,
                              (long)sizeof(info_rec),
                              seek_key,
                              status);

            check_status(status);
            printf("record updated\n");
        }/* end if */

/* prompt for next record to be updated */
        printf("What number record would you like to update");
        printf(" (1 - %d) (ctl z to stop):\n", no_of_recs);

    } /* end while */
}
/*********************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}
```

A.25. STREAM_WRITE_TAPE.C

```
/* PROGRAM stream_write_tape.c */

/* This program creates a magtape descriptor file */
/* and accesses thru STREAM calls                 */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/mts.ins.c"

typedef
    char string[80];

/* $CREATE_DEFAULT_DESC variables */
status_$t           status;
name_$pname_t       pathname;
short               namelength;
mts_$handle_t       handle;

/* $CLOSE variables */
boolean             update;

/* $GET_ATTR variables */
mts_$attr_value_t  value;

/* STREAM_$OPEN variables */
stream_$id_t        stream_id;

/* $PUT_REC variables */
string              buffer;
string             *buf_pointer;
long                buf_length;
stream_$sk_t        seek_key;

main()
{
/*load CRE_DEF_DESC variables*/

    printf("Input a new descriptor file pathname:\n");
    gets(pathname);
    namelength = strlen(pathname);

    handle = mts_$create_default_desc (pathname,
                                       namelength,
                                       status);
    check_status(status);

/* turn off the newline handling */
    value.b = false;
    mts_$set_attr(handle,
                  mts_$ascii_nl_a,
                  value,
                  status);
    check_status(status);
```

```
/*indicate an update parameter*/
    update = true; /* modify */

    mts_$close_desc( handle,
                     update,
                     status);
    check_status(status);

/* open the first tape file */

    stream_$open (pathname,
                  namelength,
                  stream_$write,              /* access      */
                  stream_$controlled_sharing, /* concurrency */
                  stream_id,
                  status);

    check_status(status);

/* write to the tape file */
    write_to_tape_file();

/* close first tape file  w/ $CLOSE */

    stream_$close (stream_id,
                   status);
    check_status(status);

/***********************************************************************/

/* change the tape file number*/

/* open the descriptor file for modification */
    handle = mts_$open_desc (pathname,
                             namelength,
                             mts_$write, /*access*/
                             status);
    check_status(status);

/* get the current file number */

    mts_$get_attr (handle,
                   mts_$file_sequence_a, /*file number*/
                   value,
                   status);
    check_status(status);

/* increment the tape file number */
    (value.i)++;

/* set new file number */

    mts_$set_attr (handle,
                   mts_$file_sequence_a, /*file number*/
                   value,
                   status);
    check_status(status);
```

```
/*close the descriptor file w/ modifications */
    update = true;

    mts_$close_desc (handle,
                     update,
                     status);
    check_status(status);

/***********************************************************************/
/* open the second tape file */

    stream_$open (pathname,
                  namelength,
                  stream_$write,              /*access*/
                  stream_$controlled_sharing, /*concurrency*/
                  stream_id,
                  status);

    check_status(status);

/* write to the tape file */
    write_to_tape_file();

/*close second tape file  w/ $CLOSE*/

    stream_$close (stream_id,
                   status);
    check_status(status);
}/* end main */

/*******************************/
write_to_tape_file()
{
/* get the input */
    printf("Input data\n");
    gets(buffer);
    while(true)
    {
/*write to file w/ $PUT_REC*/
        buf_length = strlen(buffer);
        buf_pointer = (string *)&buffer;

        stream_$put_rec (stream_id,
                         buf_pointer,
                         buf_length,
                         seek_key,
                         status);

        check_status(status);

        printf("\nInput data\n");
        if(gets(buffer) == NULL)
            break;

    }/* end while */
}/* end write_to_tape_file */
```

```
/****************************************************/
check_status(status)  /* for error handling */

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status */
```

A.26. STREAM_READ_TAPE.C

```
/* PROGRAM stream_read_tape.c */

/* This program opens a magtape descriptor file */
/* and reads thru STREAM calls                  */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/mts.ins.c"

typedef
    char string[80];

/* $OPEN_DESC variables */
status_$t          status;
name_$pname_t      pathname;
short              namelength;
mts_$handle_t      handle;

/* $CLOSE_DESC variables */
boolean            update;

/* $OPEN variables */
stream_$id_t       stream_id;

/* $GET_ATTR variables */
mts_$attr_value_t  value;

/* GET variables */
string             buffer;
string            *buf_pointer;
short              buf_length;
string            *ret_pointer;
long               ret_length;
stream_$sk_t       seek_key;
short              get;
short              number_of_recs;
char               c;
/*****************************************************/
main()
{
   printf("Input the descriptor file pathname:\n");
   gets(pathname);
   namelength = strlen(pathname);
/**************************************************************************/

/* set the file number to the beginning of the file */

/* open the descriptor file for reset */
    handle = mts_$open_desc (pathname,
                             namelength,
                             mts_$write, /* access */
                             status);
    check_status(status);
```

```
/* set file number to 1 */
    mts_$set_attr (handle,
                   mts_$file_sequence_a, /* file number */
                   1L,                   /* value       */
                   status);

    check_status(status);

/*close the descriptor file w/ modifications */
    update = true;

    mts_$close_desc (handle,
                     update,
                     status);

    check_status(status);

/**************************************************************************/

/*open the first tape file*/

    stream_$open (pathname,
                  namelength,
                  stream_$read,               /* access      */
                  stream_$controlled_sharing, /* concurrency */
                  stream_id,
                  status);

    check_status(status);

/* read from the tape file*/
    read_from_tape_file();

/* close file w/ $CLOSE */
    stream_$close( stream_id,
                   status);

    check_status(status);

/**************************************************************************/

/*advance the tape file number*/

/* open the descriptor file for modification */
    handle = mts_$open_desc (pathname,
                             namelength,
                             mts_$write, /* access */
                             status);
    check_status(status);

/* get the current file number */

    mts_$get_attr (handle,
                   mts_$file_sequence_a, /* file number */
                   value,
                   status);

    check_status(status);
```

```
/* increment the tape file number */
    (value.i)++;

/* set new file number */
    mts_$set_attr (handle,
                   mts_$file_sequence_a, /*file number*/
                   value,
                   status);

    check_status(status);

/*close the descriptor file w/ modifications */
    update = true;

    mts_$close_desc (handle,
                     update,
                     status);
    check_status(status);

/************************************************************************/

/*open the second tape file*/

    stream_$open (pathname,
                  namelength,
                  stream_$read,                 /* access      */
                  stream_$controlled_sharing, /* concurrency */
                  stream_id,
                  status);

    check_status(status);

/* read from the tape file*/
    read_from_tape_file();

/* close file w/ $CLOSE */
    stream_$close (stream_id,
                   status);

    check_status(status);
}/* end main */
/****************************************************/
read_from_tape_file()   /* for reading from tape files */
{
    printf("Input the number of records to read\n");
    scanf("%hd",&number_of_recs);
    while((c = getchar()) != '\n');

/* get records */
    buf_pointer = (string *)&buffer;
    buf_length = 256;
```

```
    for(get = 1; get<= number_of_recs; get++)
    {
        stream_$get_rec (stream_id,
                         buf_pointer,
                         buf_length,
                         ret_pointer,
                         ret_length,
                         seek_key,
                         status);

        check_status(status);

/* write the record to std output */
        printf ("%.*s\n\n", ret_length, *ret_pointer );

    }/* end for */
}/* end read_from_tape_file() */

/*****************************************************/
check_status(status)  /* for error handling */

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status */
```

A.27. STREAM_SIO_ACCESS.C

```
/* PROGRAM stream_sio_access.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/sio.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"

/* $OPEN variables */
status_$t    status;
name_$pname_t    pathname;
short    namelength;
stream_$id_t    stream_id;

/* SIO_$ variables */
sio_$value_t    value;

/*********************************************/
main()
{
    printf("Input the SIO line pathname\n");
    gets(pathname);
    namelength = strlen(pathname);

    stream_$open (pathname,
                  namelength,
                  stream_$write,         /* access      */
                  stream_$no_conc_write, /* concurrency */
                  stream_id,             /* stream_id   */
                  status);

    check_status(status);

/* INQUIRE host-synch */

    sio_$inquire (stream_id,
                 sio_$host_sync, /* inquired option */
                 value,          /* returned value  */
                 status);

    check_status(status);
```

```
    printf("The host_synch value is %s\n", (value.b) ? "TRUE" : "FALSE");
    if(value.b == true)
    {
        value.b = false;     /* turn off host-synch */

        sio_$control (stream_id,
                      sio_$host_sync,
                      value,
                      status);

        check_status(status);

/* INQUIRE new host-synch */

        sio_$inquire( stream_id,
                      sio_$host_sync, /* option */
                      value,
                      status);

        if(status.all != status_$ok )
            error_$print(status);

        printf("The host_synch value has been changed to: %s\n",
              (value.b) ? "TRUE" : "FALSE");
    }/* end if */
}/* end main */
/**********************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
    {    error_$print(status);
        pgm_$exit();
    }
}
```

A.28. STREAM_MBX_CLIENT.C

```
/* PROGRAM stream_mbx_client.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"

status_$t       status;
stream_$id_t    stream_id;
stream_$sk_t    seek_key;
short           buffer;
short          *retptr;
long            retlen;
short           i;

main()
{
/* Open the mailbox. */

    stream_$open ( "mailbox",
                   (short)7,
                   stream_$append,      /* access      */
                   stream_$unregulated, /* concurrency */
                   stream_id,
                   status);

    check_status(status);

/* Transmit some data. */
    for(i=1; i<=3; i++)
    {
       buffer = i;
       printf("Sending %d\n", buffer);

       stream_$put_rec ( stream_id,
                        &buffer,
                         (long)sizeof(buffer),
                         seek_key,
                         status);

       check_status(status);
    }/* end for */

/* Make the client wait with an open channel */

    stream_$get_rec ( stream_id,
                     &buffer,
                      (long)sizeof(buffer),
                      retptr,
                      retlen,
                      seek_key,
                      status);

     check_status(status);
```

```
/* Close the channel. */
     stream_$close( stream_id,
                    status);

     check_status(status);
}/* end main */

check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}
```

A.29. STREAM_LIST_LINKS.C

```
/* PROGRAM stream_list_links.c */

/* reads a directory and extracts entry names using streams */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"

status_$t              status;
name_$pname_t          dir_name;
short                  namelength;
stream_$id_t           stream_id;
stream_$sk_t           seek_key;
stream_$dir_entry_t    buffer;
stream_$dir_entry_t   *retptr;
long                   retlen;
/*********************************************************************/
main()
{
    printf("For which directory do you wish to list links?\n");
    gets(dir_name);

    namelength = strlen(dir_name);

    stream_$open( dir_name,
                  namelength,
                  stream_$read,
                  stream_$unregulated,
                  stream_id,
                  status);

    printf("The links in %s are;\n", dir_name);

    while(status.all == status_$ok)  /* while there is input */
    {
       stream_$get_rec ( stream_id,
                        &buffer,
                         stream_$dir_entry_size,
                         retptr,
                         retlen,
                         seek_key,
                         status);
/* test for EOF */
       if((status.code == stream_$end_of_file) &&
           (status.subsys == stream_$subs))
           break;

       check_status(status); /* test for other errors */
```

```
/* test for link - write the name */
         if(retptr->enttype == stream_$dir_enttype_link)
             printf("%.*s\n",retptr->entlen, retptr->entname );
    }/* end while */

}/* end main */

check_status(status)
/*************************************************************************/
status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}/* end check_status */
```

A.30. PAD_MAKE_WINDOWS.C

```
/* PROGRAM pad_make_windows.c */

/* This program makes an original transcript pad and window, and */
/* associates other window panes with it.                         */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define display_unit     (short)1
#define window_count     1
#define auto_close       true

stream_$id_t        source_stream;
stream_$id_t        input_stream;
stream_$id_t        edit_stream;
stream_$sk_t        seek_key;
pad_$window_desc_t  window;
pad_$window_list_t  window_list;
short               window_size;
status_$t           status;
name_$pname_t       pathname;
short               namelength;
short               count;


/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */
/* ==  Internal Procedure ============================================ */

/* This internal procedure is added only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */

hold_display()
{
time_$clock_t    rel_time;
status_$t        status;

    cal_$sec_to_clock (3L, rel_time);    /* Convert secs to UTC value */
```

```
    time_$wait ( time_$relative,          /* Time to wait                */
                 rel_time,
                 status);

    check_status(status);
}/* end hold_display() */

/* == MAIN PROGRAM ================================================== */
main()
{
/* Set position of future window.  */

    window.top    = 150;
    window.left   = 150;
    window.width  = 450;
    window.height = 450;

/* Create original transcript pad and window. */

    pad_$create_window( (char *)NULL,      /* No pathname for transcript pad   */
                        (short)0,          /* No namelength for transcript pad */
                        pad_$transcript,   /* Type of pad                      */
                        display_unit,      /* Number of display unit           */
                        window,            /* Position of window               */
                        source_stream,     /* Returns stream id                */
                        status);           /* Completion status                */

    check_status(status);

/* Close window when stream closes. */

    pad_$set_auto_close ( source_stream,  /* Stream id            */
                          window_count,   /* Number of window     */
                          auto_close,     /* Flag -- set to true  */
                          status);        /* Completion status    */
    check_status(status);

/* Make an edit pane at the bottom of the window. */

    pad_$create ( (char *)NULL,  /* Null pathname for input window        */
                  (short)0,      /* Null namelength                       */
                  pad_$input,    /* Type of pad                           */
                  source_stream, /* Same stream id as window              */
                  pad_$bottom,   /* New pane position on original pad     */
                  (short)0,      /* Pane height relative to original pad  */
                  (short)20,     /* Height maximum is 20% of original pad */
                  input_stream,  /* Returns stream id of window pane      */
                  status );      /* Completion status                     */

    check_status(status);

/* Get pathname from keyboard and set values of pathname, namelength. */
    printf("Type in the pathname of the file:\n");
    gets(pathname);
    namelength = (short)strlen(pathname);
```

```
/* Make an edit pane for the rest of the window below */
/* the frame and associate it with specified file.    */

    pad_$create ( pathname,
                  namelength,
                  pad_$edit,
                  source_stream, /* Same stream id as window   */
                  pad_$top,      /* New pane position on original pad */
                  (short)0,      /* Pane height relative to original pad */
                  (short)60,     /* Hght: 60% of original pad minus input pad */
                  edit_stream,   /* Returns stream id of window pane */
                  status );

    check_status(status);

/* Close edit pad when stream closes. */

    pad_$set_auto_close( edit_stream,
                         window_count,
                         auto_close,
                         status);

    check_status(status);
    hold_display();

/* Close the streams. */
    stream_$close( edit_stream,
                   status );
    check_status(status);

    stream_$close( input_stream,
                   status );
    check_status(status);

    stream_$close( source_stream,
                   status );
    check_status(status);
}
```

A.31. PAD_INQ_WINDOW_SIZE.C

```
/* PROGRAM pad_inq_window_size.c */

/* This program gets information about size of windows to open pad. */


#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"
#list

#define max_windows     10
#define font_size       (char *)NULL  /* No need for font pathname */

pad_$window_list_t  window_info;
short               n_windows;
short               width_scale;
short               height_scale;
pad_$string_t       font_name;
short               font_len;
short               bottom, right;
status_$t           status;
short               i;

/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

main()
{
/* Set scale to 1,1 to get width and height in raster units. */

    pad_$set_scale ( stream_$stdout,
                     (short)1,
                     (short)1,
                     status);

    check_status(status);

/* Get window information about user's standard output stream. */

    pad_$inq_windows (stream_$stdout, /* Standard output (display)      */
                      window_info,    /* Current position of window     */
                      max_windows,    /* Maximum no. of windows desired */
                      n_windows,      /* Number of windows open to pad  */
                      status);        /* Completion status              */


    check_status(status);
```

```
    printf("\n  ========================================== \n");
    if(n_windows == 1)
        printf(" One window is open to this pad.\n\n");
    else
        printf(" There are %d windows open to this pad.\n\n", n_windows);

/* Write window information for each window open to current pad. */
    for(i=0; i<n_windows; i++)
    {   bottom = window_info[i].top + window_info[i].height;
        right = window_info[i].left + window_info[i].width;
        printf(" Window %d\n", i+1);
        printf("----------\n\n");
        printf(" Upper left corner is at position");
        printf("  (%d,%d)\n", window_info[i].left, window_info[i].top);
        printf(" Lower right corner is at position");
        printf("  (%d,%d)\n", right, bottom);
        printf
        (" Width of window = %d (raster units)\n", window_info[i].width);
        printf
        (" Height of window = %d (raster units)\n\n", window_info[i].height);
    }
    printf("  ========================================== \n");
}
```

A.32. PAD_FULL_WINDOW_SHOW.C

```
/* PROGRAM pad_full_window_show.c */

/* This program uses PAD calls to manipulate full windows. */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/error.ins.c"
#list

#define no_border  false

stream_$id_t        stream_one;
status_$t           status;
pad_$window_desc_t  window;
pad_$window_list_t  windowlist;
short               winlistsize;
short               window_no;
pad_$window_desc_t  newwindow;

/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

/* ==  Internal Procedure ============================================ */

/* This internal procedure is added only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */

hold_display()
{
time_$clock_t    rel_time;
status_$t        status;

    cal_$sec_to_clock (3L, rel_time);    /* Convert secs to UTC value  */

    time_$wait (time_$relative,          /* Time to wait               */
                rel_time,
                status);

    check_status(status);
}/* end hold_display() */
```

```
/* == MAIN PROGRAM  =========================================== */
main()
{

/* Set original position of windows. */
    window.top    =  25;
    window.left   = 600;
    window.width  = 600;
    window.height = 300;

    pad_$create_window ( (char*)NULL,
                         (short)0,
                         pad_$transcript,
                         stream_$stdout,
                         window,
                         stream_one,
                         status );

    check_status(status);
    hold_display();

    pad_$inq_windows ( stream_one,
                       windowlist,
                       winlistsize,
                       window_no,
                       status );

    check_status(status);

    pad_$make_invisible( stream_one,
                          window_no,
                          status );

    check_status(status);

    pad_$inq_full_window ( stream_one,
                           window_no,
                           newwindow,
                           status );

    check_status(status);

    pad_$set_full_window ( stream_one,
                           window_no,
                           newwindow,
                           status );

    check_status(status);
    hold_display();

    pad_$make_invisible ( stream_one,
                          window_no,
                          status );

    check_status(status);
}
```

A.33. PAD_WINDOW_SHOW.C

```
/* PROGRAM pad_window_show */

/* This program shows how to pop and push windows, */
/* make a window visible and invisible, and remove */
/* a window border.                                */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"

#define display_unit  (short)1
#define auto_close    true
#define no_border     false
#define pop           true
#define push          false

stream_$id_t        stream_one;
stream_$id_t        stream_two;
stream_$id_t        stream_three;
stream_$id_t        pane_stream;
pad_$window_desc_t  window_one;
pad_$window_desc_t  window_two;
pad_$window_desc_t  window_three;
short               window_no1;
short               window_no2;
short               window_no3;
pad_$window_list_t  window_list;
status_$t           status;

/* ==  INTERNAL PROCEDURE ========================================== */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

/* == MAIN PROGRAM  ================================================ */
main()
{
/* Set the original positions of the windows. */

    window_one.top    = 10;
    window_one.left   = 550;
    window_one.width  = 300;
    window_one.height = 300;

    window_two.top    = 350;
    window_two.left   = 175;
    window_two.width  = 300;
    window_two.height = 300;
```

```
    window_three.top    = 300;
    window_three.left   = 300;
    window_three.width  = 300;
    window_three.height = 300;

/* Create three transcript pads with full windows. */

    pad_$create_window( (char *)NULL,
                        (short)0,
                        pad_$transcript,
                        display_unit,
                        window_one,
                        stream_one,
                        status);

    check_status(status);

    pad_$create_window( (char *)NULL,
                        (short)0,
                        pad_$transcript,
                        display_unit,
                        window_two,
                        stream_two,
                        status);

    check_status(status);


    pad_$create_window( (char *)NULL,
                        (short)0,
                        pad_$transcript,
                        display_unit,
                        window_three,
                        stream_three,
                        status);

    check_status(status);

/* Get value of window_no1 window_no2 and window_no3 for next calls. */

    pad_$inq_windows( stream_one,   /* Stream id                  */
                      window_list,  /* Location, size of window   */
                      (short)0,     /* No need for position info  */
                      window_no1,   /* Returns no of windows      */
                      status );     /* Status code                */

    check_status(status);

    pad_$inq_windows( stream_two,   /* Stream id                  */
                      window_list,  /* Location, size of window   */
                      (short)0,     /* No need for position info  */
                      window_no2,   /* Returns no of windows      */
                      status );     /* Status code                */

    check_status(status);
```

```
    pad_$inq_windows( stream_three, /* Stream id                 */
                      window_list,  /* Location, size of window  */
                      (short)0,     /* No need for position info */
                      window_no3,   /* Returns no of windows     */
                      status );     /* Status code               */

    check_status(status);

/* Make windows close when stream closes. */

    pad_$set_auto_close( stream_one,
                         window_no1,
                         auto_close,
                         status);

    check_status(status);

    pad_$set_auto_close( stream_two,
                         window_no2,
                         auto_close,
                         status);

    check_status(status);

    pad_$set_auto_close( stream_three,
                         window_no3,
                         auto_close,
                         status);

    check_status(status);

/* Remove border from the last window. */

    pad_$set_border( stream_three,
                     window_no3,
                     no_border,
                     status);

    check_status;

/* Push the last window open to the bottom. */

    pad_$pop_push_window( stream_three,
                          window_no3,
                          push,
                          status);

    check_status(status);

/* Pop the last window open to the top. */

    pad_$pop_push_window( stream_three,
                          window_no3,
                          pop,
                          status);
    check_status(status);
```

```
/* Make the second window invisible. */

    pad_$make_invisible( stream_two,
                         window_no2,
                         status);

    check_status(status);

/* Make the first window invisible. */

    pad_$make_invisible( stream_one,
                         window_no1,
                         status);

    check_status(status);

/* Make the first window visible again. */

    pad_$select_window( stream_one,
                        window_no1,
                        status);

    check_status(status);

/* Create pad on borderless window, note that */
/* in doing so, the border re-appears.        */

    pad_$create( (char *)NULL,
                 (short)0,
                 pad_$input,
                 stream_three,
                 pad_$bottom,
                 (short)0,
                 (short)20,
                 pane_stream,
                 status);

    check_status(status);

/* Close streams before terminating program. */

    stream_$close( stream_one,
                   status);

    check_status(status);

    stream_$close( stream_two,
                   status);

    check_status(status);

    stream_$close( stream_three,
                   status);

    check_status(status);
}
```

A.34. PAD_INQ_DISP_KBD.C

```
/* PROGRAM pad_inq_disp_kbd.c */

/*This program gets information about the user's display type and keyboard. */


#nolist
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/vfmt.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define max_windows     10
#define font_size       0
#define buffer          256

status_$t           status;
pad_$display_type_t display_type;
short               unit_number;
pad_$string_t       kbd_suffix;
short               suffix_length;

/* == INTERNAL PROCEDURE ====================================== */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */
/* ==  Internal Procedure =========================================== */

/* This internal procedure is added only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */

hold_display()
{
time_$clock_t    rel_time;

    cal_$sec_to_clock (3L, rel_time);      /* Convert secs to UTC value */

    time_$wait ( time_$relative,           /* Time to wait              */
                 rel_time,
                 status);

    check_status(status);
}/* end hold_display() */
```

```
/* == MAIN PROGRAM =========================================== */
main()
{
    pad_$inq_disp_type ( stream_$stdout, /* Standard output stream - display */
                         display_type,   /* Returns type of display          */
                         unit_number,    /* Returns unit number, always 1    */
                         status);        /* Completion status                */

    check_status(status);

    printf("\n  ============================================= \n");
    printf("    Number of display units: \n\n");

    if(unit_number == 1)
        printf(" There is one display unit connected to this node.\n ");
    else
        printf(" %d display units are connected to this node\n",unit_number);
    printf("  ============================================= \n");
    printf(" Type of display: \n\n");
    switch(display_type)
    {   case pad_$bw_15P:
            printf(" This is a black and white portrait. \n");
            break;
        case pad_$bw_19L:
            printf(" This is a black and white landscape. \n");
            break;
        case pad_$color_display:
            printf(" This is a color display (1024 x 1024 pixels).\n");
            break;
        case pad_$800_color:
            printf("This is a color display ( 1024 x 800 pixels).\n");
            break;
        case pad_$none:
            printf(" There is no display. \n");
            break;
        default:
            printf("ERROR - UNKNOWN DISPLAY TYPE\n");
            break;
    }/* end switch */

/*  Find out which keyboard is in use. */

    pad_$inq_kbd ( stream_$stdout,   /* Standard output stream         */
                   buffer,           /* Size of string buffer          */
                   kbd_suffix,       /* Returns keyboard suffix string */
                   suffix_length,    /* Returns keyboard suffix length */
                   status );         /* Completion status              */

    check_status(status);
    printf("  ============================================= \n");
```

```
    if(suffix_length == 0)
    {   printf("The keyboard sufffix is 0\n");
        printf("User has the low-profile keyboard.\n");
    }
    else
    {
        if(kbd_suffix[suffix_length - 1] == '2')
        {   printf("Keyboard suffix is \"%.*s\"\n",suffix_length,kbd_suffix);
            printf("\n User has the 880 keyboard.\n");
        }
        else
            printf(" Not sure which keyboard is in use.\n");
    }/* end else */
/* Redefine the keyboard function keys. */
    if((suffix_length == 0) || (kbd_suffix[suffix_length - 1] == '2'))
    {   printf("\n Redefining low-profile function keys. \n");

        pad_$def_pfk ( stream_$stdout,    /* Stream id                    */
                      "F1   ",            /* Keyname                      */
                      "TT   ",            /* DM command -- to top of window */
                       (short)2,          /* Length of DM command         */
                       status );

        check_status(status);
        hold_display();
    }/* end if */
    printf("\n  ============================================== \n");
}/*end main() */
```

A.35. PAD _ SCALE.C

```
/* PROGRAM pad_scale.c */

/* This program is a sample of using PAD_$SET_SCALE.                  */
/* It creates window one with a transcript pad that is 5 lines high.*/
/* The second window created has a transcript pad that is 20 raster */
/* units high.                                                      */

#include <stdio.h>
#include </sys/ins/base.ins.c>
#include </sys/ins/pad.ins.c>
#include </sys/ins/error.ins.c>
#include </sys/ins/streams.ins.c>
#include </sys/ins/pgm.ins.c>
#include </sys/ins/cal.ins.c>
#include </sys/ins/time.ins.c>

#define display_unit  (short)1
#define auto_close    true
#define window_no     (short)1

stream_$sk_t        seek_key;
stream_$id_t        stream_one;
stream_$id_t        stream_two;
stream_$id_t        pane_stream_one;
stream_$id_t        pane_stream_two;
status_$t           status;
pad_$window_desc_t  window_one;
pad_$window_desc_t  window_two;
pad_$window_list_t  window_list;   /* Array of up to 10 windows      */
short               window_size;   /* Maximum no. of windows desired */

/* ==  INTERNAL PROCEDURE ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

/* ==  INTERNAL PROCEDURE ============================================ */

/* This internal procedure is used only to demonstrate this program.   */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you     */
/* can see how each call works.                                        */


hold_display()
{
time_$clock_t    rel_time;

   cal_$sec_to_clock (3L,
                      rel_time);     /* Convert secs to UTC value */
```

```
    time_$wait ( time_$relative,     /* Time to wait                */
                 rel_time,
                 status);

    check_status(status);

}/* end hold_display() */
/* == MAIN PROGRAM  ================================================ */
main()
{
    window_one.top    = 100;
    window_one.left   = 100;
    window_one.width  = 300;
    window_one.height = 300;

    window_two.top    = 550;
    window_two.left   = 550;
    window_two.width  = 300;
    window_two.height = 300;

/* = ONE ==================================================== */

/* Open the window as a transcript pad. */

    pad_$create_window ( (char *)NULL,
                         (short)0,
                         pad_$transcript,
                         display_unit,
                         window_one,
                         stream_one,
                         status);
    check_status(status);

    pad_$create ( (char *)NULL,
                  (short)0,
                  pad_$transcript,
                  stream_one,
                  pad_$top,
                  pad_$abs_size,      /* Pad is absolute value */
                  (short)5,           /* 5 lines high          */
                  pane_stream_one,
                  status);

    check_status(status);

/* = TWO ==================================================== */

/* Open the window as a transcript pad. */

    pad_$create_window ( (char *)NULL,
                         (short)0,
                         pad_$transcript,
                         display_unit,
                         window_two,
                         stream_two,
                         status);
```

```
    check_status(status);

/* Set scale of window two height and width to be in raster units. */

    pad_$set_scale ( stream_two,
                     (short)1,
                     (short)1,
                     status);

    check_status(status);

    pad_$create ( (char *)NULL,
                  (short)0,
                  pad_$transcript,
                  stream_two,
                  pad_$top,
                  pad_$abs_size,       /* Pad absolute size */
                  (short)20,           /* Raster units      */
                  pane_stream_two,
                  status);
    check_status(status);

    hold_display();

    pad_$set_auto_close ( stream_two,
                          window_no,
                          auto_close,
                          status);

    check_status(status);


    pad_$set_auto_close ( stream_one,
                          window_no,
                          auto_close,
                          status);

    check_status(status);

    stream_$close( stream_one,
                   status);

    check_status(status);

    stream_$close( stream_two,
                   status);

    check_status(status);
}
```

A.36. PAD_INQ_FONT.C

```
/* PROGRAM pad_inq_font.c                                              */

/* This program is a sample of using PAD_$SET_SCALE, and PAD_$INQ_FONT. */
/* It creates a window and frame and writes the prompt, #, within the   */
/* frame. It uses PAD_$INQ_FONT to find out how high to make the frame. */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"

#define display_unit   (short)1
#define max_windows    1
#define prompt_str     "# "
#define auto_close     true


stream_$sk_t        seek_key;
stream_$id_t        pane_stream;
status_$t           status;
pad_$window_list_t  window_info;
short               n_windows;
short               font_len;
short               font_height;
short               font_width;

/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */
/* ==  Internal Procedure ============================================ */

/* This internal procedure is added only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */

hold_display()
{
time_$clock_t    rel_time;

    cal_$sec_to_clock (3L, rel_time);    /* Convert secs to UTC value  */

    time_$wait ( time_$relative,         /* Time to wait               */
                 rel_time,
                 status);

    check_status(status);
}/* end hold_display() */
```

```
/* == MAIN PROGRAM ================================================= */
main()
{
/* Get the size of the current window. */

    pad_$inq_windows ( stream_$stdout,
                       window_info,      /* Current position of window */
                       max_windows,      /* Maximum no. of windows desired */
                       n_windows,        /* Number of windows open to pad */
                       status );
    check_status(status);

/* Get the width and height of current font. */

    pad_$inq_font ( stream_$stdout,
                    font_width,
                    font_height,
                    (char *)NULL,    /* No need to know name */
                    (short)0,        /* No need to know name */
                    font_len,
                    status );
    check_status(status);

/* Set scale of window height and width to raster units. */

    pad_$set_scale ( stream_$stdout,
                     (short)1,
                     (short)1,
                     status);
    check_status(status);

    pad_$create_frame ( stream_$stdout,
                        window_info[0].width, /* Same size as window        */
                        font_height,          /* Same height as font height */
                        status );
    check_status(status);

    pad_$move ( stream_$stdout,
                pad_$absolute,
                (short)5,             /* Raster units   */
                font_height,          /* Height of font */
                status );
    check_status(status);

/* Put the prompt, #, in the input window with STREAM_$PUT_CHR. */

    stream_$put_chr ( stream_$stdout,
                      &prompt_str,         /* Pointer to buffer       */
               (long)strlen(prompt_str),   /* Number of bytes to read */
                       seek_key,
                       status );

    check_status(status);
    hold_display();

    stream_$close( stream_$stdout,
                   status);
    check_status(status);
}
```

A.37. PAD_DIGCLK.C

```
/* PROGRAM pad_digclk.c */

/* This program displays a digital clock on the screen. The user          */
/* executes the program with the DM command, CPO.  The user can optionally */
/* and the x,y coordinates on the command line to specify its location.    */
/* Otherwise the clock runs in the top left corner of the screen.          */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/vfmt.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/pfm.ins.c"
#list

#define font_name      "f9x15.iv"   /* Font file located in /sys/dm/fonts */
#define window_num      1
#define as_time_len     8
#define border_size     5
#define close           true
#define no_border       false

status_$t            status;
pad_$window_desc_t   window = { 0, 0, 10, 10 }; /* default window location */
stream_$id_t         stream;
short                font_id;
short                font_height;
short                font_width;
short                hunoz;
short                hukairz;
time_$clock_t        one_second = {0,0,0,250000};
cal_$timedate_rec_t  now;
short                last_minute = -1;
char                 as_time[as_time_len];  /* ascii time */
stream_$sk_t         key;

/* == INTERNAL PROCEDURE =============================================== */
/*   Status checking internal procedure */

check_status(status)
status_$t status;
{   if(status.all != status_$ok)
        pfm_$error_trap(status);
}
/* == INTERNAL PROCEDURE  ============================================== */
/* Internal procedure to see if user provided cursor location. */
/* PGM_$GET_ARG returns a string, so convert it to an integer. */
/* If all goes well, the result is assigned to arg_val.        */

get_num_arg(arg_num, addr_arg_val)
short arg_num;
short *addr_arg_val;
```

```
{
char    arg[80];
short    argl;
short    hunoz;
short    hukairz;
short    anyway;
short    number;

/* Get argument from command line and assign its length to argl. */

    argl = pgm_$get_arg ( arg_num,       /* Number of argument      */
                          arg,           /* Returns argument string */
                          status,        /* Completion status       */
                   (short)sizeof(arg) ); /* Max length of argument  */


    if(status.all == status_$ok)
    {

/* Convert string to integer and assign to variable, hunoz */
        hunoz = vfmt_$decode2( "%wd%.",  /* String                  */
                               arg,      /* Text buffer             */
                               argl,     /* Size of text buffer     */
                               hukairz,  /* Number of fields decoded */
                               status,   /* Completion status       */
                               number,   /* Decoded data            */
                               anyway );/* Decoded data            */

        if(status.all == status_$ok)
            *addr_arg_val = number;
    }/* end if */
}/* end get_num_arg() */
/* == MAIN PROGRAM  ==================================================== */
main()
{
/* Get window left coordinate, if user supplies it. */
    get_num_arg( 1, &(window.left) );

/* Get window top coordinate, if user supplies it.  */
    get_num_arg( 2, &(window.top) );

/* Create the window -- note that the size is 10x10 pixels, we */
/* will change it to after we know the font size.              */

    pad_$create_window ( (char *)NULL,      /* Null pathname          */
                         (short)0,          /* Null namelength        */
                         pad_$transcript,   /* Type of pad            */
                         (short)1,          /* Number of display unit */
                         window,            /* Position of window     */
                         stream,            /* Stream id              */
                         status );          /* Completion status      */
    check_status(status);

    pad_$set_auto_close( stream,
                         window_num,
                         close,
                         status );
    check_status(status);
```

```
/* Load the font, seven_seg, and use it. */

    pad_$load_font( stream,
                    font_name,
                    (short)strlen(font_name),
                    font_id,                 /* Returns font id */
                    status );
    check_status(status);

    pad_$use_font(stream, font_id, status);
    check_status(status);

/* Get the size of the font in use. */
    pad_$inq_font( stream,
                   font_width,               /* Returns width of font */
                   font_height,              /* Returns height of font */
                   hunoz,                    /* no need to know value   */
                   (char)0,                  /* No need to know font name */
                   hukairz,                  /* no need to know value   */
                   status );

    check_status(status);

/* Adjust window width and height to font size. */
    window.width  = font_width * as_time_len + border_size;
    window.height = font_height + border_size;

/* Make window borderless. */
    pad_$set_border( stream, window_num, no_border, status );
    check_status(status);

/* Set scale to pixel values. */
    pad_$set_scale( stream,
                    (short)1,
                    (short)1,
                    status );
    check_status(status);

/* Set window to new size. */
    pad_$set_full_window( stream, window_num, window, status );
    check_status(status);

/* Create a frame the same size as the window itself. */
    pad_$create_frame( stream, window.width, window.height, status );
    check_status(status);

    while(true)
    {
/* Translate a system clock value into time value. */
        cal_$decode_local_time(now);
        if( now.minute != last_minute)
        {
/* If a minute has passed, clear the frame and */
/* write the minute and second value.          */

            pad_$clear_frame( stream,
                              OL,
                              status );
            check_status(status);
```

```
            vfmt_$encode5( "%2wd:%2zwd:%2zwd%$", as_time, as_time_len,
                           hunoz, now.hour, now.minute, now.second, OL, OL );

/* Put the output cursor at the left side of the frame. */

            pad_$move( stream,
                       pad_$absolute,
                       border_size,
                       font_height,    /* Must be at least font_height */
                       status );

            check_status(status);

            stream_$put_rec( stream,
                             &as_time,
                       (long)sizeof(as_time),
                              key,
                              status);
            check_status(status);

        }/* end if */
        else
        {
/* Just write the second value. */

            vfmt_$encode2( "%2zwd%$", as_time, sizeof(as_time),
                           hunoz, now.second, OL );

/* Move the output cursor to the 6th character position. */
            pad_$move( stream,
                       pad_$absolute,
                (short)(border_size + 6 * font_width),
                       font_height,
                       status );

            check_status(status);

            stream_$put_rec( stream,
                             &as_time,
                             2L,
                             key,
                             status);
            check_status(status);
        }/* end else */

        last_minute = now.minute;

        time_$wait( time_$relative, one_second, status );
        check_status(status);
    }/* end while */
}/* end main */
```

A.38. PAD_MAKE_ICON.C

```
/* PROGRAM pad_make_icon.c */

/* This program is a sample of using icons. */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"

#define display_unit  (short)1
#define auto_close    true

stream_$id_t        stream_win;
stream_$id_t        pane_stream;
stream_$sk_t        seek_key;
status_$t           status;
pad_$window_desc_t  window;       /* Position, height, width of window */
short               window_no;    /* Number of windows open to a pad   */
pad_$window_list_t  window_list;  /* Array of up to 10 windows         */
short               window_size;  /* Maximum no. of windows desired    */
pad_$position_t     icon_pos;     /* Position of icon                  */
char                icon_char;
boolean             icon_moved = false; /* Checks if icon moved */

/* ==  Internal Procedure ================================================== */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        exit(1);
    }
}/* end check_status() */

/* ==  Internal Procedure ================================================== */

/* This internal procedure is used only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you   */
/* can see how each call works.                                      */

hold_display()
{
time_$clock_t    rel_time;

   cal_$sec_to_clock (3L, rel_time); /* Convert secs to UTC value */

    time_$wait ( time_$relative,     /* Time to wait              */
                 rel_time,
                 status);

    check_status(status);
}/* end hold_display() */
```

```
/* == MAIN PROGRAM =============================================== */
main()
{
/*  Set postion of future windows. */

     window.top    = 10;
     window.left   = 10;
     window.width  = 300;
     window.height = 300;


/*  Set position of icon to upper right corner.  */

    icon_pos.x_coord = 1020;
    icon_pos.y_coord = 24;

/* Create a new transcript window in icon format. */

    pad_$create_window ( (char *)NULL,     /* No pathname for transcript pad */
                         (short) 0,        /* No namelength                  */
                         pad_$transcript,  /* Type of pad                    */
                         display_unit,     /* Which display unit -- 1        */
                         window,           /* Location of future window      */
                         stream_win,       /* Stream id of new window        */
                         status );         /* Completion status.             */

    check_status(status);

/*  Get window statatics for next calls. */

    pad_$inq_windows ( stream_win,    /* Stream id                     */
                       window_list,   /* Location, size of window      */
                       window_size,   /* Max number of windows desired */
                       window_no,     /* Number of windows open to pad */
                       status );      /* Status code                   */

    check_status(status);

/* Make window close when stream closes. */

    pad_$set_auto_close ( stream_win,
                          window_no,
                          auto_close,
                          status );

    check_status(status);

/* Do work with window ... */

/* Change window into an icon. */
    icon_char = ' ';

  pad_$make_icon ( stream_win,  /* Stream id                */
                   window_no,   /* Window number            */
                   icon_char,   /* Default character icon */
                   status );    /* Completion status        */

  check_status(status);
  hold_display();
```

```
/* Move position of icon and change the icon character. */
   icon_pos.x_coord = 950;
   icon_pos.y_coord = 710;
   icon_char        = '*';

  pad_$set_icon_pos ( stream_win, /* Stream id         */
                      window_no,  /* Window number     */
                      icon_pos,   /* Position of icon  */
                      icon_char,  /* Icon character    */
                      status );   /* Completion status */
  check_status(status);

/* Suspend process until user expands window from icon format. */

    pad_$icon_wait ( stream_win,
                     window_no,
                     icon_moved,  /* TRUE if icon moved           */
                     icon_pos,    /* Returns new position of icon */
                     status );
    check_status(status);
    hold_display();

/* Turn transcript window into an icon. */

    pad_$make_icon ( stream_win,
                     window_no,
                     icon_char,
                     status );

    check_status(status);
    hold_display();

    stream_$close ( stream_win,
                    status );

    check_status(status);
}/*end main() */
```

A.39. PAD_CREATE_ICON.C

```
/* PROGRAM pad_create_icon.c */

/* This is sample program on using icons. */

/* It creates a new window with an input pad in icon format.         */
/* It uses STREAM_$PUT_CHR to put a prompt in the input window, and  */
/* STREAM_$GET_REC to get input from the keyboard.                   */

/* Note the internal procedure, HOLD_DISPLAY uses a TIME_$WAIT call to  */
/* suspend the program's activity, so that someone running this program */
/* can see how the different stages work.                               */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/time.ins.c"


#define display_unit    1
#define prompt_str      "# "
#define auto_close      true

stream_$id_t        stream_win;
stream_$id_t        pane_stream;
stream_$sk_t        seek_key;
status_$t           status;
pad_$window_desc_t  window;      /* Position, height, width of window */
short               window_no;   /* Number of windows open to a pad   */
pad_$window_list_t  window_list; /* Array of up to 10 windows         */
short               window_size; /* Maximum no. of windows desired    */
pad_$position_t     icon_pos;    /* Position of icon                  */
char                icon_char;
boolean             icon_moved;  /* Indicates whether user moved icon */
char                buffer[80];  /* Buffer to hold keyboard input     */
char               *return_ptr[80];
long                return_len;

/* == INTERNAL PROCEDURE  ========================================= */
check_status(status)   /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         exit(1);
    }
}
/* ==  INTERNAL PROCEDURE  ======================================== */

/* This internal procedure is used only to demonstrate this program.  */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */
```

```
hold_display()
{
time_$clock_t  rel_time;

    cal_$sec_to_clock( 3L, rel_time); /* Convert secs to UTC value */

    time_$wait( time_$relative,
                rel_time,
                status );   /* Time to wait */

   check_status(status);
}/* end hold_display */

/* == MAIN PROGRAM  ================================================ */
main()
{
/*  Set postion of future windows. */

    window.top    = 10;
    window.left   = 10;
    window.width  = 300;
    window.height = 300;


/*  Set position of icon to upper right corner.  */

    icon_pos.x_coord = 1020;
    icon_pos.y_coord = 24;

/* Create a new transcript window in icon format.  The icon will have */
/* the shell icon character from the default icon font.              */

    icon_char = '*';

    pad_$create_icon ( (char)0,          /* No pathname for transcript pad  */
                       (short)0,         /* No namelength                   */
                       pad_$transcript,  /* Type of pad                     */
                       display_unit,     /* Which display unit -- 1         */
                       icon_pos,         /* Location -- x and y coordinates */
                       icon_char,        /* Icon font displayed             */
                       window,           /* Location of future window       */
                       stream_win,       /* Stream id of new window         */
                       status );         /* Completion status.              */
    check_status(status);

/* Create an input pad for the new transcript pad. This is a */
/* window pane associated with the same window.              */

    pad_$create((char*)0,    /* No pathname for input pad                    */
                (short)0,    /* No namelength for input pad                  */
                pad_$input,  /* Type of pad                                  */
                stream_win,  /* Stream id of related transcript pad          */
                pad_$bottom, /* Input pads always go on bottom               */
                (short)0,    /* Next argument, size parameter is relative */
                (short)20,   /* New pad takes up 20% of related window       */
                pane_stream, /* Stream id of this input pad                  */
                status );    /* Completion status                            */

    check_status(status);
```

```
/*  Get window statatics for next calls. */

    pad_$inq_windows ( stream_win,     /* Stream id                     */
                       window_list,    /* Location, size of window      */
                       window_size,    /* Max number of windows desired */
                       window_no,      /* Number of windows open to pad */
                       status );       /* Status code                   */

    check_status(status);

/* Make window close when stream closes. */

    pad_$set_auto_close ( stream_win,
                          window_no,
                          auto_close,
                          status );
   check_status(status);

/* Suspend process until user opens icon. It checks to see if icon */
/* has moved,  if it has, it moves the icon to the new position    */
/* when it returns to an icon.                                     */

    printf("                      ***                            \n");
    printf("Process suspended until user turns icon into window,\n");
    printf( "or until user moves the icon. If user turns icon into\n" );
    printf( "a window, it waits for input.  After user types input,\n" );
    printf( "it waits 3 seconds, then turns the window into an icon.\n" );
    printf ( "                      ***                            \n" );

    pad_$icon_wait ( stream_win,
                     window_no,
                     icon_moved,        /* TRUE if icon moved.           */
                     icon_pos,          /* If TRUE, new position of icon. */
                     status );

   check_status(status);

/* Put the prompt, #, in the input window with STREAM_$PUT_CHR. */

    stream_$put_chr ( stream_win,          /* Stream of transcript pad */
                      &prompt_str,         /* Pointer to buffer        */
                (long)strlen(prompt_str),  /* Number of bytes to read  */
                      seek_key,
                      status );
    check_status(status);

/* Get information from input pad with stream_$get_rec. */

    stream_$get_rec ( pane_stream,
                      &buffer,        /* Buffer holding input*/
                (long)sizeof(buffer),
                      return_ptr,     /* Return pointer    */
                      return_len,     /* Return length     */
                      seek_key,       /* Seek key          */
                      status );       /* Completion status */

   check_status(status);

/* After user turns icon into window, this     */
```

```
/* program returns it to an icon in 5 seconds. */

    hold_display();   /* Internal procedure to suspend action. */

/* Turn transcript window into an icon. */

    pad_$make_icon ( stream_win,
                     window_no,
                     icon_char,
                     status );

    check_status(status);
    hold_display();

/* Now, program turns window from icon format to full-sized window. */

    printf("The program will now automatically turn the window\n");
    printf ("from icon format to full-sized window and then terminate.\n");

    pad_$select_window ( stream_win,
                         window_no,
                         status );

    check_status(status);
    hold_display();

    stream_$close ( stream_win,
                    status );

    check_status(status);
}/* end main */
```

A.40. PAD_FILENAME.C

```
/* PROGRAM pad_filename.c */

/* This program will display a file and print the filename at the top.*/

/* First it creates a new pad and a window to view the file.  It then */
/* creates a transcript pad and window pane to hold frame.  It creates */
/* frame to hold the filename, and lastly it creates an edit pad for  */
/* the rest of the pad.     */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/vfmt.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define display_unit   1
#define window_count   (short)1
#define pane_size      (short)1
#define max_frame_size 32767

typedef
    char bufstring[512];    /* String buffer */

stream_$id_t        source_stream;
stream_$id_t        pane_stream;
stream_$id_t        pane_edit_stream;
stream_$sk_t        seek_key;

pad_$window_desc_t  window;
pad_$window_list_t  window_list;
short               window_size;
short               frame_width;
short               frame_height;

status_$t           status;
name_$pname_t       pathname;
short               namelength;
short               count;

static short        source_name_font = -1;
pad_$string_t       inverted_font_name;  /* Buffer to make inverted name. */
short               font_height;
short               font_width;
short               font_len;
pad_$string_t       font_name;           /* Size of font returned         */
short               i;
boolean             auto_close = true;
```

```
/* == FUNCTION ============================================================ */

/* This function copies a given string to a buffer, and returns the number  */
/* of characters to be copied.  It stops when at the character pair '%$'.   */

stringcopy( src, addr_dst)
bufstring src;
bufstring *addr_dst;
{
short    i, j;    /* Indexes to src and dst strings */

    i = 0;                   /* Initialize the indexes  */
    j = 0;
    while((src[i] != '%') || (src[i+1] != '$'))
        (*addr_dst)[j++] = src[i++];
    return(j); /* The number of characters copied. */
}

/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

/* ==  Internal Procedure ============================================ */

/* This internal procedure is added only to demonstrate this program. */
/* It calls TIME_$WAIT to suspend the process for 3 seconds so you    */
/* can see how each call works.                                       */

hold_display()
{
time_$clock_t    rel_time;
status_$t        status;

   cal_$sec_to_clock (3L, rel_time);     /* Convert secs to UTC value */

    time_$wait (time_$relative,         /* Time to wait              */
                rel_time,
                status);

    check_status(status);
}/* end hold_display() */

main()
{
/* Set position of future window.  */
    window.top    = 10;
    window.left   = 10;
    window.width  = 500;
    window.height = 500;
```

```
/* Get pathname from keyboard and set values of pathname, namelength. */
    printf("Type in the pathname of the file: \n");
    gets(pathname);
    namelength = (short)strlen(pathname);

/* Create original transcript pad and window. */

    pad_$create_window((char *)NULL,     /* No pathname for transcript pad   */
                       (short)0,         /* No namelength for transcript pad */
                       pad_$transcript,  /* Type of pad                      */
                       display_unit,     /* Number of display unit           */
                       window,           /* Position of window               */
                       source_stream,    /* Returns stream id                */
                       status);          /* Completion status                */

    check_status(status);

/* Close window when stream closes. */

    pad_$set_auto_close(source_stream,  /* Stream id            */
                        window_count,   /* Number of window     */
                        auto_close,     /* Flag -- set to true */
                        status);        /* Completion status    */

    check_status(status);

/* Make a transcript pad and window pane for the name of file. */

    pad_$create((char *)NULL,    /* No pathname                    */
                (short)0,        /* No namelength                  */
                pad_$transcript, /* Type of pad                    */
                source_stream,   /* Same stream id as window above */
                pad_$top,        /* Location of new window pane    */
                pad_$abs_size,   /* Pane size is asbsolute value   */
                pane_size,       /* Pane height is 1 line          */
                pane_stream,     /* Stream id of window pane       */
                status );        /* Completion status              */

    check_status(status);

/* Close window when stream closes. */

    pad_$set_auto_close(pane_stream,
                        window_count,
                        auto_close,
                        status);
    check_status(status);

/* Now make frame in above pad to hold inverted pathname. */

    frame_width  = max_frame_size;
    frame_height = pane_size;

    pad_$create_frame(pane_stream,    /* Same as window pane */
                      frame_width,    /* Same as window pane */
                      frame_height,   /* Same as window pane */
                      status );
```

```
    check_status(status);

/* Before printing the filename, find out the  */
/* inverted font name of the font name in use. */

    strcpy(inverted_font_name, " ");

    pad_$inq_font(source_stream,    /* Stream id of original transcript pad */
                  font_width,       /* Returns width of font                */
                  font_height,      /* Returns height of font               */
                  font_name,        /* Returns name of font                 */
           (short)sizeof(font_name),/* Size of buffer for font_name         */
                  font_len,         /* Length of font_name                  */
                  status );         /* Completion status                    */

    strcpy(inverted_font_name, font_name);

/* Assume font is not bold, try loading the bold inverted   */
/* version of the same font by adding the extension (.b.iv) */
/* to the font_name with the function, stringcopy.          */
    i = font_len + stringcopy(".b.iv%$", inverted_font_name + font_len);

    pad_$load_font(pane_stream,         /* Stream of frame     */
                   inverted_font_name,  /* Font_name + .b.iv   */
                   i,                   /* Length of font_name */
                   source_name_font,    /* Returns ID of font  */
                   status);             /* Completion status   */

/* If the font is already bold, it returns an error, so */
/* try adding the inverted extension (.iv) only.        */

        if(status.all != status_$ok)
        {   i = font_len + stringcopy (".iv%$", inverted_font_name + font_len);

            pad_$load_font(pane_stream,
                           inverted_font_name,
                           i,
                           source_name_font,
                           status);

            if(status.all != status_$ok)
            {   source_name_font = 0;    /* Use the default font. */
                status.all = status_$ok;
            }
        }

/* Now clear the frame to erase any old filenames, and write the new name. */

    pad_$clear_frame(pane_stream,
                     0L,         /* Clear entire frame */
                     status );

    check_status(status);
```

```
/* Use pad_use_font to have program use the desired font. */
    if(source_name_font != 0)
    {   pad_$use_font(pane_stream,        /* Stream of frame */
                      source_name_font,   /* Font id returned above */
                      status );

        status.all = status_$ok;
    }

/* Put output cursor in frame. */
    pad_$move(pane_stream,
              pad_$absolute,  /* Move relative to top left of frame */
              (short)5,       /* x-coordinate relative to frame     */
              (short)1,       /* y coordinate relative to frame     */
              status);

    check_status(status);

/* Write name of file in frame. */
    vfmt_$ws2(pane_stream,
              "%A%.",
              pathname,
              namelength);

    hold_display();

/* Make an edit pane for the rest of the window below */
/* the frame and associate it with specified file.    */

    pad_$create(pathname,
                namelength,
                pad_$edit,
                source_stream,    /* Same stream id as window                */
                pad_$bottom,      /* New pane position on original pad       */
                (short)0,         /* Pane height relative to original pad    */
                (short)100,       /* Height = 100% of original pad minus frame.*/
                pane_edit_stream, /* Returns stream id of window pane        */
                status );

    check_status(status);

/* Close edit pad when stream closes. */

    pad_$set_auto_close(pane_edit_stream,
                        window_count,
                        auto_close,
                        status);
    check_status(status);
```

```
/* Close the streams. */

    stream_$close(pane_edit_stream,
                  status);

    check_status(status);

    stream_$close(pane_stream,
                  status);

    check_status(status);

    stream_$close(source_stream,
                  status);

    check_status(status);
}/* end main */
```

A.41. PAD_RAW_MODE.C

```
/* PROGRAM pad_raw_mode.c */

/* This program shows how to use raw mode. It asks for your   */
/* password but does not echo the input to the screen.  After */
/* you type in your password, it replies,"Thank you."         */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/pad.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"

#define display_unit   (short)1
#define auto_close     true
#define message        " Enter your password: "
#define reply          " Thank you. "
#define window_no     1

stream_$id_t        stream_one;
stream_$id_t        pane_stream;
stream_$sk_t        seek_key;
status_$t           status;
pad_$window_desc_t  window_one;
pad_$window_list_t  window_list;
short               move_char;
char                buffer[80]; /* Buffer to hold keyboard input */
char               *return_ptr;
long                return_len;
short               i;

/* ==  Internal Procedure ============================================ */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}/* end check_status() */

/* == MAIN PROGRAM  ================================================ */
main()
{   window_one.top    = 10;
    window_one.left   = 550;
    window_one.width  = 300;
    window_one.height = 300;

/* Create an original transcript pad and a window. */

    pad_$create_window ((char*)NULL,
                        (short)0,
                        pad_$transcript,
                        display_unit,
                        window_one,
                        stream_one,
                        status);
```

```
    check_status(status);

/* Make windows close when stream closes. */

    pad_$set_auto_close(stream_one,
                        window_no,
                        auto_close,
                        status);

    check_status(status);

/* Create an input pad and initialize it in raw mode. */

     pad_$create ((char *)NULL,
                  (short)0,
                  pad_$input,
                  stream_one,
                  pad_$bottom,
                  pad_$init_raw,
                  (short)20,
                  pane_stream,
                  status);

    check_status(status);

/* Write message in window. */

     stream_$put_rec ( stream_one,
                      &message,
                       (long)strlen(message),
                       seek_key,
                       status);

    check_status(status);

/* Get input from keyboard.  It gets each        */
/* character until it reaches a carriage return. */
    i=0;
    do
    {
        stream_$get_rec( pane_stream,         /* Standard input -- keyboard */
                        &buffer[i],           /* Buffer holding input       */
                   (long)sizeof(buffer) -i + 1,
                         return_ptr,          /* Return pointer             */
                         return_len,          /* Return length              */
                         seek_key,            /* Seek key                   */
                         status );            /* Completion status          */

        check_status(status);
        i += return_len;
     }while(buffer[i - 1] != pad_$cr);
```

```
/* Move output cursor to where the message text ends. */

    move_char = sizeof(message) + 1;

    pad_$move( stream_one,
               pad_$absolute,
               move_char,
               (short)1,
               status);

    check_status(status);

/* Write reply in window. */

    stream_$put_rec ( stream_one,
                      &reply,
                 (long)strlen(reply),
                      seek_key,
                      status);
    check_status(status);

/* Return to normal cooked processing. */

    pad_$cooked ( pane_stream,
                  status );

    check_status(status);

    stream_$close( stream_one,
                   status);

    check_status(status);
}
```

A.42. PAD_PASTE_BUFFER.C

```
/* PROGRAM pbufs_paste_buffer.c */

/* This program manipulates paste buffers. It asks for user to supply the name */
/* of the paste buffer. If it exists, it writes the contents of the buffer. */
/* If it does not exist, it reads lines of input from the keyboard until the */
/* user types CTRL/Z.  It repeats the sequence, asking the  user to supply */
/* names of paste buffers until the user types STOP. */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"
#include "/sys/ins/name.ins.c"
#include "/sys/ins/pbufs.ins.c"
#include "/sys/ins/pad.ins.c"

#define text  true

stream_$id_t    stream_buf;
status_$t       status;
name_$pname_t   info;
name_$pname_t   buffer_name;
short           buffer_namelen;
stream_$sk_t    seek_key;
long            buflen;
name_$pname_t  *retptr;
long            retlen;
boolean         done;
short           i;

/* == INTERNAL PROCEDURE  ========================================== */
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
      error_$print(status);
}

/* == INTERNAL PROCEDURE  ========================================== */
error_routine()
{
    pgm_$set_severity(pgm_$error);
    pgm_$exit;
}

/* == MAIN PROGRAM  ================================================ */
main()
{

/* Write initial prompt */
```

```
    done = false;
    printf("          =================             \n");
    printf(" Type the name of the paste buffer (Or type STOP to quit.):\n\n ");
    gets(buffer_name);
    if((! strcmp(buffer_name, "stop")) || (! strcmp(buffer_name, "STOP")))
        done = true;
    while(!done)
    {

/* pbufs_open needs blank padded array so store name length and pad w/ blanks */
        buffer_namelen = strlen(buffer_name);
        for(i=0; buffer_name[i++];);          /* find the null terminator */
        buffer_name[--i] = ' ';               /* pad with blanks          */
        while( i < sizeof(buffer_name) -1)
            buffer_name[i++] = ' ';

/* buffer_name is no longer null terminated at the end of the string. */
/* When printing buffer_name, specify a maximum precision or padded */
/* blanks will be printed.          */

        buffer_name[i] = '\0';                /* just in case.            */



/* Open existing paste buffer and write contents to screen. */

        pbufs_$open ( buffer_name, /* Name of existing buffer */
                      text,        /* Text buffer             */
                      stream_buf,  /* Returns stream id       */
                      status );    /* Completion status       */

        if(status.all == status_$ok)
        {

/* Read data from existing paste buffer. */
            printf("          =================             \n");
            printf
("This is the contents of paste buffer %.*s:\n\n", buffer_namelen, buffer_name);

            while(status.all == status_$ok)
            /* Read a line and write it to the screen. */
            {
                 stream_$get_rec( stream_buf,    /* Stream id                 */
                                 &info,          /* Address of input line     */
                            (long)strlen(info),  /* Length of input line      */
                                  retptr,        /* Returns pointer to input  */
                                  retlen,        /* Returns length of input   */
                                  seek_key,      /* Seek key                  */
                                  status);       /* Completion status         */

                if(status.code == stream_$end_of_file)  /* Test for EOF */
                    break;
                (*retptr)[retlen] = '\0';    /* null terminate returned buffer */
                printf( "   %s\n", *retptr);/* Write buffer line to screen     */

                if(status.all != status_$ok)
                    error_routine();
            }/* while there is input */
        }/* if */
```

```
        else
        {   if(status.code == stream_$name_not_found)

 /* Input data in new paste buffer */
            {
                pbufs_$create ( buffer_name, /* Name of buffer              */
                                text,        /* Text buffer                 */
                                stream_buf,  /* Returns stream id of buffer */
                                status );    /* Completion status           */

                check_status(status);

/* Get information from keyboard for paste buffer */
                printf("         =================             \n\n");
                printf(" Type information for paste buffer,
                         one line at a time.");
                printf(" (Or type CTRL/Z to stop.):\n" );

                while(gets(info) != NULL)   /* User has input */
                {   buflen = strlen(info);

/* Terminate line with NEWLINE character */
                    info[buflen] = '\n';

                    stream_$put_rec ( stream_buf, /* Stream id              */
                                     &info,       /* Address of input line */
                                      buflen,     /* Length of input line  */
                                      seek_key,   /* Seek key              */
                                      status);    /* Completion status     */

                    check_status(status);
                    printf("\n Type another line, or CTRL/Z to stop. ");
                }/* while */

                printf("         =================             \n");
                printf(" Information is now in the paste buffer: %.*s\n\n",
                       buffer_namelen, buffer_name );
            }/* if */
            else
                printf(" Cannot read or write to paste buffer.\n");
        }/* end else */

/* Repeat prompt */
        printf("         =================             \n");
        printf(" Type the name of the paste buffer: ");
        printf(" (Or type STOP to quit.) \n");
        gets(buffer_name);
        if((! strcmp(buffer_name, "stop")) || (! strcmp(buffer_name, "STOP")))
            done = true;
    } /* WHILE NOT done */
}
```

A.43. EC_TIME_KBD_EVENTS.C

```
/* PROGRAM ec_time_kbd_events.c */

/* This program waits for keyboard input and time.  If user types input,*/
/* it echoes the input.  It writes out the date and time every 10       */
/* seconds. The program continues until the user types CTRL/Z.          */

#include <stdio.h>
#include  "/sys/ins/base.ins.c"
#include  "/sys/ins/streams.ins.c"
#include  "/sys/ins/ec2.ins.c"
#include  "/sys/ins/time.ins.c"
#include  "/sys/ins/cal.ins.c"
#include  "/sys/ins/vfmt.ins.c"
#include  "/sys/ins/error.ins.c"
#include  "/sys/ins/pgm.ins.c"

/* Define indexes for arrays */
#define kbd_ec    0   /* Fist element in array for keyboard events. */
#define time_ec   1   /* Second element in array for time events.   */


ec2_$ptr_t            ec2_ptr[2]; /* Array of pointers to two       */
                                  /* eventcounts.  First element    */
                                  /* points to keyboard eventcount; */
                                  /* second element points to time  */
                                  /* eventcount.                    */

long                  ec2_val[2]; /* Array of eventcount trigger       */
                                  /* values.  First element is trigger */
                                  /* for keyboard event; second is     */
                                  /* trigger for time event.           */

stream_$id_t          sio_strm;   /* Stream id */
status_$t             status;     /* Status code */
stream_$sk_t          seek_key;   /* Seek key */
short                 which;      /* Number returned by EC2_$WAIT */
char                  line[80];
char                 *linep[80];
long                  linelen;
time_$clock_t         clock;
cal_$timedate_rec_t   d_clock;
short                 dummy;


/* == MAIN  ======================================================== */
main()
{
    printf("\nThis program uses eventcounts to wait for keyboard input.\n");
    printf("If you type a line of input, it will echo the input.\n");
    printf("It will also write the date and time every 10 seconds.\n\n");
    printf("\nType input or CTRL/Z to quit.\n\n");

/* Get the EC for standard input to the keyboard.  Store the      */
/* eventcount pointer in the first element of the pointer array. */
```

```
    stream_$get_ec(stream_$stdin,         /* Stream id                  */
                   stream_$getrec_ec_key, /* Type of eventcount         */
                   ec2_ptr[kbd_ec],       /* Returns Eventcount pointer */
                   status);               /* Completion status          */

    check_status(status);

/* Get the EC for time.  Store the eventcount pointer */
/* in the second element of the pointer array.        */

    time_$get_ec (time_$clockh_key, /* Time key                            */
                  ec2_ptr[time_ec], /* Eventcount pointer returned by call */
                  status);          /* Completion status                   */

     check_status(status);

/* Prime the eventcount trigger values. Read the value of the   */
/* keyboard EC and store it in the first element of the trigger */
/* value array.  Read the value of the time EC and store it in  */
/* the second element of the trigger value array.               */

    ec2_val[kbd_ec] = ec2_$read (*(ec2_ptr[kbd_ec]));
    ec2_val[time_ec] = ec2_$read (*(ec2_ptr[time_ec]));

/* Go into an infinite loop to wait for input from the two sources. */
/* The first time through the loop, both eventcounts are satisfied. */
    do
    {
/* Determine which event count reaches satisfaction first. */

        which = ec2_$wait (ec2_ptr,  /* EC pointers           */
                           ec2_val,  /* EC triggers           */
                           (short)2, /* Number of eventcounts */
                           status);  /* Completion status     */

        check_status(status);

/* decrement which to adjust for initial array index of 0 in C */
        which--;

        switch(which)
        {   case kbd_ec:  /* If WHICH is 0, enter keyboard loop */
                do
                {

/* Read the current eventcount, increment it, and save it as the new trigger. */
                    ec2_val[kbd_ec] = ec2_$read(*(ec2_ptr[kbd_ec])) + 1;

/* Get keyboard input until no more data, then return to outer loop. */

                    stream_$get_conditional (stream_$stdin, /* Stream id */
                                             &line,   /* Buffer to read line */
                                       (long)sizeof(line),  /* Bufferlen */
                                             linep,/* Pointer to returned data */
                                             linelen, /* Length of data */
                                             seek_key,/* Seek key */
                                             status); /* Completion status */
                    check_status(status);
```

```
                    if(linelen > 0)
                    {   (*linep)[linelen] = '\0';
                        printf("* Keyboard Input: * %s\n", *linep);
                        printf("\nType input or CTRL/Z to quit.\n\n");
                    }
                }while(linelen != 0);/* end do */
                break;

            case time_ec:

/* Read the current eventcount, increment it  */
/*10 seconds, and save it as the new trigger. */

            ec2_val[time_ec] = ec2_$read(*(ec2_ptr[time_ec])) + 40;

/* Get time clock and print it. */
            time_$clock(clock);
            cal_$decode_local_time(d_clock);

            vfmt_$write10
            ("* Date: * %2WD/%2WD/%4WD  %2X * Time: * %2ZWD:%2ZWD:%2WD %.",
                        d_clock.month,
                        d_clock.day,
                        d_clock.year,
                        d_clock.hour,
                        d_clock.minute,
                        d_clock.second,
                        dummy, dummy, dummy, dummy);
            break;
        }/* end switch */

/* repeat until user types CTRL/Z */
    }while((status.code != stream_$end_of_file) &&
           (status.subsys != stream_$subs)); /* end do */
}/* end main */

/* == INTERNAL PROCEDURE  ========================================== */

check_status(status)
status_$t status;
{   if(status.all != status_$ok)
        if((status.code != stream_$end_of_file) &&
           (status.subsys != stream_$subs))
        {   error_$print(status);
            exit(1);
        }
}/* end check_status() */
```

A.44. EC_WAIT_FOR_TIME.C

```
/* PROGRAM ec_wait_for_time.c */

/* This program inhibits asynchronous faults from occuring while waiting for */
/* input.  If no input occurs within 20 seconds, the time eventcount will    */
/* be satisfied, and the program will enable asynchronous faults.            */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/ec2.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/pgm.ins.c"

#define kbd_ec     0
#define time_ec    1

ec2_$ptr_t    ec2_ptr[2];
long          ec2_val[2];
short         which;
status_$t     status;
stream_$sk_t  seek_key;
char          line[80]; /*return buffer*/
char         *linep[80];
long          linelen;
char          name[80];
boolean       time_out;
/* == MAIN ======================================================== */
main()
{
    printf("This program requests a program name.  If you do not\n");
    printf("supply one in 20 seconds, the program will terminate.\n");
    printf("Faults are inhibited while the program waits for input.\n\n");

/* Get an eventcount to wait for input from standard in (usually the kbd!) */

    stream_$get_ec (stream_$stdin,          /* stream-id              */
                    stream_$getrec_ec_key,  /* stream-key             */
                    ec2_ptr[kbd_ec],        /* returned pointer to ec */
                    status);

    check_status(status);

/* Get a time eventcount to wait an amount of time */

    time_$get_ec (time_$clockh_key,    /* time-key               */
                  ec2_ptr[time_ec],    /* returned pointer to ec */
                  status);

    check_status(status);
```

```
/* Prime the eventcount trigger values, except the time eventcount. */

    ec2_val[kbd_ec] = ec2_$read(*(ec2_ptr[kbd_ec]));

/* Immediately advance the satisfaction value -  20 sec.*/

    ec2_val[time_ec] = ec2_$read (*(ec2_ptr[time_ec])) + 80;

    time_out = false;

/* Disable CTRL/Q sequence while waiting for input or */
/* until time limit reached. */
    pfm_$inhibit();
    printf("Faults inhibited.\n");
    while(!time_out)
    {
/* Determine which event count reaches satisfaction first. */
/* You force  all eventcounts to be satisfied except time. */

        which = ec2_$wait (ec2_ptr,    /* ec pointer array */
                           ec2_val,    /* ec value array   */
                           (short)2,   /* number of ec's   */
                           status);

        check_status(status);

/* decrement which to adjust for initial array index of 0 in C */
        which--;
        switch(which)
        {   case kbd_ec:  /* This value is reached first...*/

/* Immediately advance the satisfaction value*/
                ec2_val[kbd_ec] = ec2_$read (*(ec2_ptr[kbd_ec])) + 1;

/* get keyboard input */

                stream_$get_conditional(stream_$stdin, /* stream-id */
                                       &line,      /* pointer to buffer */
                                  (long)sizeof(line), /* # of bytes requested */
                                        linep,   /* returned pointer to buffer */
                                        linelen, /* returned length off buffer */
                                        seek_key,
                                        status);
                check_status(status);
                if(linelen > 0)
                {   (*linep)[linelen] = '\0';
                    strcpy(name, *linep);
                    printf("Input File name: %s", name);
                    time_out = true;
                }
                break;

            case time_ec: /* If the time ec value is satisfied first... */
```

```
/* immediately advance the satisfaction value --  20 sec. */
                ec2_val[time_ec] = ec2_$read(*(ec2_ptr[time_ec])) + 80;
                pfm_$enable();   /* OK to interrupt now. */
                time_out = true;
                printf("No action for 20 seconds.\n");
                break;
        }/* end switch */
    }/* end while */
    pfm_$enable();
    printf("Faults enabled.\n");
    printf("Terminating program.\n");
}/*end main */
/***************************************************************************/
check_status(status)   /* for error handling */

status_$t    status;
{   if(status.all != status_$ok)
    {    error_$print(status);
         exit(1);
    }
}
```

A.45. CAL_DECODE_LOCAL.C

```
/* PROGRAM cal_decode_local.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/cal.ins.c"

main()
{
/* This program  decodes local time and prints it */

cal_$timedate_rec_t  d_clock;

    /* get decoded local time */
    cal_$decode_local_time (d_clock);

    /* write it to the screen */


    printf ("date: %2d/%2d/%4d time : %2d:%2d:%2d\n",
             d_clock.month,
             d_clock.day,
             d_clock.year,
             d_clock.hour,
             d_clock.minute,
             d_clock.second);

}
```

A.46. TIME_ZONE.C

```
/* PROGRAM time_zone.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/error.ins.c"

/* ==  Internal Procedure ================================================== */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        exit(1);
    }
}/* end check_status() */

/* == MAIN PROGRAM  ================================================ */
main()
{
/* This program gets local time zone information (offset & name)*/
/* determines the TZ difference in min. by name and by hr:min   */

    status_$t status;

    /* GET_INFO variables */
    cal_$timezone_rec_t tz_info   ;

    /* TZDIF variables */
    char           time_zone[4];
    short          tzn_length;
    short          tz_dif;
    cal_$tz_name_t tz_name;
}
```

```
    /* get offset using timezone name */
    printf ("What time_zone do you want the difference of? ");
    scanf ("%s", time_zone);
    tzn_length = strlen(time_zone);
    cal_$decode_ascii_tzdif (time_zone,
                             tzn_length,
                             tz_dif,
                             tz_name,
                             status);
    check_status (status);

    /* write timezone offset to screen */
    printf ("The time zone offset is: %4d\n", tz_dif);

    /* get timezone offset using time difference */

    printf ("Input time difference ( [+|-] HR:MIN ): \n");
    scanf ("%s", time_zone);
    tzn_length = strlen(time_zone);
    cal_$decode_ascii_tzdif (time_zone,
                             tzn_length,
                             tz_dif,
                             tz_name,
                             status);
    check_status (status);

    /* write timezone offset to screen */
    printf ("The time zone offset is: %4d\n", tz_dif);

}
```

A.47. CAL_ADD_TIMES.C

```
/* PROGRAM cal_add_times.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/cal.ins.c"

main()
{
linteger                seconds;
time_$clock_t           rel_time,
                        clock1;
cal_$timedate_rec_t     d_clock;

    /* input number of seconds to add to time */
    printf("How many seconds to add? \n");
    scanf("%d", &seconds);

    /* convert number of seconds to internal value */
    cal_$sec_to_clock(seconds,    /* # of seconds    */
                      rel_time);  /* internal format */

    /* get local time */
    cal_$get_local_time(clock1);

    /* add the times */
    cal_$add_clock(clock1,    /* in/out */
                   rel_time);

    /* get the result in readable form */
    cal_$decode_time(clock1,    /* internal format */
                     d_clock); /* readable format */

    /* write the result to the screen */
    printf("time resulting from add: %02d:%02d:%02d\n",
            d_clock.hour,
            d_clock.minute,
            d_clock.second);
}
```

A.48. CAL_SUB_TIMES.C

```
/* PROGRAM cal_sub_times.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/error.ins.c"

/* ==  Internal Procedure ================================================== */
check_status(status)
status_$t status;
{   if(status.all != status_$ok)
    {   error_$print(status);
        exit(1);
    }
}/* end check_status() */

/* == MAIN PROGRAM  ================================================ */
main()
{
status_$t             status;
char                  date[8],
                      time[8];
cal_$timedate_rec_t c_clock;
time_$clock_t         clock,
                      curr_time;
boolean               sub_check;
long                  num_of_sec;


    /* get the input */
    printf ("Enter date to subtract ( yr/month/day )? \n");
    scanf  ("%s",date);

    /* convert ASCII string to system readable date */
    cal_$decode_ascii_date (date,
                            8,
                            c_clock.year,
                            c_clock.month,
                            c_clock.day,
                            status);
    check_status(status);

    /* get the input */
    printf ("Enter time (hr:min:sec - 24 hr format)? \n");
    scanf  ("%s",time);

    cal_$decode_ascii_time (time,
                            8,
                            c_clock.hour,
                            c_clock.minute,
                            c_clock.second,
                            status);
    check_status(status);
```

```
   /* convert readable format to internal format */
   cal_$encode_time (c_clock,
                     clock);

   /* get local time */
   cal_$get_local_time (curr_time);

   /* subtract input time from the current time */
   sub_check = cal_$sub_clock (curr_time,
                               clock);

   /* convert difference to seconds */
   num_of_sec = cal_$clock_to_sec (curr_time);

   /* check if result is negative - print error */
   if (sub_check != true)
      printf ("Subtraction resulted in negative value \n");
   else
      printf ("Seconds difference %d\n", num_of_sec);
}
```

A.49. TIME_COMPARE.C

```
/* PROGRAM time_compare.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/type_uids.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/pgm.ins.c"

#define SETSIZE (short)32

status_$t                status;
name_$pname_t            pathname1, pathname2;
short                    namelength1, namelength2;

/* INQUIRE variables */
stream_$inquire_mask_t  input_mask;
stream_$ir_opt           inquiry_type;
stream_$ir_rec_t         attributes;
stream_$inquire_mask_t  error_mask;

/* time variables */
time_$clock_t            time1, time2;
time_$clock_t            most_recent_time;
short                    cmp_check;
cal_$timedate_rec_t      decoded_time;


main()
{
/* get the first pathname -- and its length */
    printf("Input the first pathname:\n");
    gets(pathname1);
    namelength1 = strlen(pathname1);

/* get the second pathname -- and its length */
    printf("Input the second pathname\n");
    gets(pathname2);
    namelength2 = strlen(pathname2);

/* initialize inquire variables */
    lib_$init_set(input_mask, SETSIZE);

/* date/time modified           */
    lib_$add_to_set(input_mask, SETSIZE, (short)stream_$dtm);

 /* get by name even if not open */
    inquiry_type = stream_$name_unconditional;
    strcpy(attributes.obj_name , pathname1);
    attributes.obj_namlen = namelength1;
```

```
/* get date/time modified on pathname1 */
    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);

    check_status(status);
    time1.high = attributes.dtm;
    time1.low  = 0;

/* get date/time modified on pathname2 */
    strcpy(attributes.obj_name, pathname2);
    attributes.obj_namlen = namelength2;

    stream_$inquire (input_mask,
                     inquiry_type,
                     attributes,
                     error_mask,
                     status);

    check_status(status);

    time2.high = attributes.dtm;
    time2.low  = 0;

/* compare times and assign most_recent_time */
    cmp_check = cal_$cmp_clock ( time1,
                                 time2 );

    printf("Cmp_check: %d\n", cmp_check);

    switch(cmp_check)
    {

    case  0 : /* times are equal */
        printf("%s and %s  are the same age.\n", pathname1, pathname2);
        most_recent_time = time1;   /* could be time2 -- no difference */
        break;

    case 1 :  /* 1 is older than 2 */
        printf("%s is newer than %s.\n", pathname1, pathname2);
        most_recent_time = time1;
        break;

    case -1 :   /* 2 is older than 1 */
        printf("%s is newer than %s.\n", pathname2, pathname1);
        most_recent_time = time2;
        break;

    default :
        printf("ERROR -- BAD RETURN VALUE FROM CAL_$CMP_CLOCK\n");
    }/* switch */
```

```
/* decode most recent dtm */
    cal_$apply_local_offset(most_recent_time);

    cal_$decode_time( most_recent_time,
                      decoded_time);

    printf("DATE/TIME MODIFIED: ");
    printf("%d/%d",decoded_time.month ,decoded_time.day);
    printf("/%d", (decoded_time.year % 100));
    printf("   %d:%d", decoded_time.hour, decoded_time.minute);
    printf(":%d\n", decoded_time.second);
}/* end main */

check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_$exit();
    }
}
```

A.50. TIME_WAIT_ABS.C

```
/* PROGRAM time_wait_abs.c */

/* This program waits an absolute time to send a reminder */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pgm.ins.c"

status_$t               status;
char                    reminder[80];

/* DATE and TIME variables */
char                    date[80];
char                    time[80];

/* ENCODE, WAIT variables  */
cal_$timedate_rec_t  c_clock;
time_$clock_t        abs_time;
time_$clock_t        curr_time;
boolean              sub_check;
linteger             num_of_sec;

main()
{
/*input the reminder text*/
    printf("Input reminder text:\n");
    gets(reminder);

/*get the input*/
    printf("When do you wish to be reminded?\n");
    printf ("Date: ( yr/month/day )?:\n");
    gets(date);

    cal_$decode_ascii_date (date,
                             (short)8, /* date length */
                             c_clock.year,
                             c_clock.month,
                             c_clock.day,
                             status);

    check_status(status);

/* get the input */
    printf("Time: (hr:min:sec - 24 hr format)? \n");
    gets(time);

    cal_$decode_ascii_time (time,
                             (short)8,      /* time length */
                             c_clock.hour,
                             c_clock.minute,
                             c_clock.second,
                             status);
```

```
   check_status(status);

/*Convert TIMEDATE_REC_T to TIME_$CLOCK*/

   cal_$encode_time( c_clock,   /* input  */
                     abs_time); /* result */

/* remove local offset to a time_$clock_t */
   cal_$remove_local_offset (abs_time);

/* WAIT for an ABSOLUTE time */
   time_$wait (time_$absolute,   /* pre-defined        */
               abs_time,         /* time to wait until */
               status);

   check_status(status);

   printf("%s\n",reminder);
}
/*****************************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
    {   error_$print(status);
        pgm_exit();
    }
}
```

A.51. TIME_WAIT_OR_DEFAULT.C

```
/* PROGRAM  time_wait_or_default.c */

#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/streams.ins.c"
#include "/sys/ins/ec2.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/cal.ins.c"
#include "/sys/ins/error.ins.c"


#define time_ec   0  /*ec array indices*/
#define kbd_ec    1

typedef
    char string[80];

status_$t     status;
ec2_$ptr_t    ec2_ptr[2];
long          ec2_val[2];
short         which;

/* GET_CONDITIONAL variables */
stream_$sk_t  seek_key;
string        line;          /*return buffer*/
string       *linep;
long          linelen;
string        name;
short         p_count;

main()
{
/* Get an eventcount to wait on for input from standard in (usually the kbd!) */

    stream_$get_ec( stream_$stdin,          /* stream-id              */
                    stream_$getrec_ec_key,  /* stream-key             */
                    ec2_ptr[kbd_ec],        /* returned pointer to ec */
                    status);

    check_status(status);

/* Get a time eventcount to wait an amount of time */

    time_$get_ec (time_$clockh_key, /* time-key               */
                  ec2_ptr[time_ec], /* returned pointer to ec */
                  status);

    check_status(status);

/* Prime the eventcount trigger values       */
/* Get the current value of both event counts */

    ec2_val[kbd_ec] = ec2_$read (*(ec2_ptr[kbd_ec]));
    ec2_val[time_ec] = ec2_$read (*(ec2_ptr[time_ec]));
```

```
/* NOW GO INTO A LOOP PROMPTING FOR INPUT */
    linelen = 0;
    p_count = 0;

    do
    {
/*determine which event count reaches satisfaction first*/
        which = ec2_$wait (ec2_ptr,     /*ec pointer array*/
                           ec2_val,     /*ec value array*/
                           (short)2,    /*number of ec's*/
                           status);
        check_status(status);

 /* decrement which to adjust for initial array index of 0 in C */
        which--;

        switch(which)
        {
        case kbd_ec:    /*if the keyboard ec value is reached first...*/

/*immediately advance the satisfaction value*/
            ec2_val[kbd_ec] = ec2_$read (*(ec2_ptr[kbd_ec])) + 1;

/*get keyoard input*/

            stream_$get_conditional (stream_$stdin, /* stream-id */
                                    &line,     /* pointer to buffer */
                               (long)sizeof(line), /* #of bytes requested */
                                     linep,    /* returned pointer to buffer */
                                     linelen,  /* returned length off buffer */
                                     seek_key,
                                     status);
            check_status(status);
            if(linelen > 0)
               strcpy(name, *linep);
            break;

/* if the time ec value is satisfied first... */
        case time_ec:

/* immediately advance the satisfaction value -  10 sec. */
            ec2_val[time_ec] = ec2_$read (*(ec2_ptr[time_ec])) + 40;

/* prompt again */
             if(p_count < 4)
                 printf("Input a program name:\n");
             else
                 printf("The default program name is being used.\n");
             break;

        }/* end switch */

/* advance the prompt count */
        p_count++;

/* repeat until input is received or 3 prompts have occured */
    }while((linelen <= 0) && (p_count != 5));
}
```

```
/************************************************/
check_status(status)

status_$t  status;
{
    if(status.all != status_$ok)
        error_$print(status);
}
```

Index

I/O
stream ID 4-3

%%, VFMT format directive 8-13
%, VFMT format directive 8-4, 8-12
%/, VFMT format directive 8-13
%A, VFMT format directive 8-2, 8-5
%D, VFMT format directive 8-2, 8-10
%dw, VFMT format directive 8-9
%E, VFMT format directive 8-7, 8-8, 8-9, 8-11, 8-12
%F, VFMT format directive 8-8
%fw, VFMT format directive 8-7, 8-9, 8-11
%H, VFMT format directive 8-2, 8-10
%J, VFMT format directive 8-9, 8-11
%K, VFMT format directive 8-8
%L, VFMT format directive 8-8, 8-10, 8-12
%M, VFMT format directive 8-7
%O, VFMT format directive 8-10
%P, VFMT format directive 8-10, 8-12
%S, VFMT format directive 8-10, 8-11
%T, VFMT format directive 8-13
%U, VFMT format directive 8-8, 8-11
%W, VFMT format directive 8-10, 8-12
%X, VFMT format directive 8-13
%Z, VFMT format directive 8-8, 8-9, 8-11

/dev directory 9-8
/dev/sio 9-8
/SYS/DM/FONTS
font files 5-23
/SYS/DM/FONTS/ICON
default icon set 5-38
/SYS/INS/ 1-1

[ ], default pane size attribute 5-6

_$ 1-1

'NODE_DATA/PASTE_BUFFERS
directory 5-57

Aborting process 5-10
and continuing program 2-17
Absolute mode for TPAD calls
definition 5-61
Absolute seeking
definition 4-31
Absolute time, definition 7-9
Absolute/Relative mode for TPAD calls
definition 5-62
Access type
compatible 4-14
definition 4-12
guidelines 4-12
Accessing
arguments from an invoked program 3-20
arguments from child process 3-20
locked system resoures 7-16
magtape files 9-12
mailboxes 9-3
objects, DOMAIN 9-1
objects, how IOS manager controls 4-12
protected memory (fault) 2-1
serial lines 9-8
ADDR, Pascal function 2-23
Address
fault reported 2-9
Address space
getting more by creating a child process 3-6
using predefined PGM_$ARG argument type 3-18
Alarm server
giving user control over the display 5-4
Argument count
definition 3-18
Argument vector
accessing argument in UNIV_PTR format 3-21
definition 3-18
deleting arguments from 3-22

index number of argument pointer 3-23
Arguments
C sample program A-20, A-22, A-23, A-24
deleting arguments when invoking a child program 3-22
passing predefined data types 1-1
passing to invoked programs 3-18
Returning all from child process 3-21
Returning from child process one at a time 3-20
See also Parameters, passing
Arrays
DOMAIN predefined data types 1-7
of eventcount pointers 6-5
of records, DOMAIN predefined 1-7
of trigger values 6-6
passing character arrays in C 1-37
ASCII
creating table of associated hexadecimal and decimal values 8-15
object type UID 9-2
See also UASC object type
ASCII character strings
building a character table with VFMT (example) 8-14
converting to system time 7-7
formatting 8-4
ASCII data
converting to hexadecimal 8-1
ASCII format
converting time to system time 7-7
Asynchronous faults 6-7
C sample program A-111
continuing program after 6-13
definition 6-13
disabled during clean-up handler 2-17
disabling with EC2_$WAIT 6-14
disabling with EC2_$WAIT_SVC 6-16
during eventcounts 6-13
error message 2-6
handling with EC2_$WAIT_SVC (example) 6-17
ignoring with PFM_$INHIBIT 6-13
inhibiting 2-12, 2-25
keeping track of inhibited 2-25
previously disabled faults 6-16, 6-17
program response when enabled, disabled 6-13
reenabling 2-16
summary of how program responds (table) 6-18
using a time eventcount with 6-14
See also Faults

Background mode
creating a new process 3-11
Background process
C sample program A-14
BACKSPACE
control character 5-3
Backup objects
creating 4-8
Based variables
referencing in FORTRAN 1-16
Binary files
object type UID 9-2
Bits
setting in C 1-31
setting the bit field 1-30
setting, in FORTRAN 1-18
testing a group of 1-20, 1-33
testing bits one-by-one 1-19, 1-32
testing in C 1-31
testing in FORTRAN 1-19
BOF 4-4
Boolean data type
as a separate data type 1-13
emulating in C 1-30
emulating in FORTRAN 1-13
in a record 1-13
Building a character table
with VFMT format directives (example) 8-14
Busy-waiting, alternative to eventcounts 6-3

C

Boolean type 1-9
clearing a bit from set in 1-34
emulating Boolean data type in 1-30
emulating large sets in 1-34
emulating predefined data types in 1-30
emulating records in 1-35
emulating set data type in 1-30
emulating variant records in 1-36
passing character arrays in 1-37
passing integer constants 1-39
passing integer paramters in 1-38
passing parameters to system calls 1-37
predefined Boolean type 1-30
referencing structure members in 1-37
setting bits in 1-31
testing bits in 1-31
testing return status of system call 2-3
unions 1-36
using predefined data types 1-4
C sample programs
emulating a large set 1-34
emulating a record 1-36
emulating status code variant record with C union 1-36
passing integer parameters 1-38
passing parameters as character arrays 1-37
setting bits in 1-31
testing a number of bits in 1-33
testing bits one-by-one 1-32
Cache
fault reported 2-10
CAL_$ADD_CLOCK 7-9, 7-10
CAL_$APPLY_LOCAL_OFFSET 7-2
CAL_$CLOCK_TO_SEC 7-9
CAL_$CMP_CLOCK 7-9, 7-13
CAL_$DECODE_ASCII_DATE 7-7
CAL_$DECODE_ASCII_TZDIF 7-4
CAL_$DECODE_ASCII_TIME 7-7
CAL_$DECODE_LOCAL_TIME 1-36, 5-33, 7-3
CAL_$DECODE_TIME 7-6
CAL_$ENCODE_TIME 7-7
CAL_$FLOAT_CLOCK 7-9
CAL_$GET_INFO 7-4
CAL_$GET_LOCAL_TIME 7-2
CAL_$REMOVE_LOCAL_OFFSET 7-4, 7-17
CAL_$SEC_TO_CLOCK 5-12, 7-10
CAL_$SUB_CLOCK 7-9, 7-11
CAL_$TIMEDATE_REC_T 1-23, 7-2, 7-3, 7-7
CAL_$TIMEZONE_REC_T 7-4
CAL_$WEEKDAY 1-17
Calendar
See also time
Carriage control characters
FORTRAN 4-16
Changing window display
specifying character fonts 5-23
CHAR
as a Boolean field in a record 1-14
Character arrays 1-37
passing in C 1-37
Character fonts
PAD system calls requiring font size 5-25
See also font files
Check_status
procedure to check call completion 5-12
Checking for input with STREAM_$GET_CONDITIONAL 6-10
Child process
C sample program A-11, A-19, A-20, A-22, A-30
communicating with parent only initially 3-17
converting to orphan process 3-17
definition 3-6
deleting arguments before invoking 3-22
getting process information about 3-29
getting streams from parent 3-6
handling process eventcount 3-8

inherits environment from parent 3-6
own address space 3-6
passing program name to program in a 3-19
permanent operations 3-7
program name 3-20
releasing resources after completed 3-7
setting a severity level 2-7
severity levels 3-7
using an eventcount to wait for (example) 3-8
waiting for 3-7
Clean-up handlers
applying to specific program section 2-14
C sample program A-6
establishing 2-12
exiting 2-15
exiting by passing a fault status 2-15
exiting by passing a severity level 2-15
handling faults with 2-12
multiple 2-15
passing control 2-12
reestablishing handler, returning to program 2-15
releasing 2-14
releasing multiple handlers 2-15
returning to the program 2-17
testing for PFM_$CLEANUP_SET 2-6
testing state before clean up 2-15
Cleaning up after asynchronous fault 6-13
Clock
C sample program A-85
Closing
frames 5-43
windows and window panes 5-10
Completion status
severity level 3-3
Concurrency mode
compatibility 4-14
definition 4-12
guidelines 4-12
Concurrent processing
performing 3-1, 3-6
waiting for a child process 3-7
Connection
See also stream connection
Connection attributes
definition 4-17
Connection vector
definition 3-24
Control code
definition 5-56
Controlling
system output with cursors 5-52
access to objects 4-12
the mouse 5-61
the touchpad 5-61
Controlling file access with eventcounts 6-1
Converting
child process to orphan process 3-17
data with VFMT system calls 8-1
internal time values to readable output 7-2
readable time to system time 7-7
relative system time to floating point value 7-9
relative system time to integer value 7-9
relative time to TIME_$CLOCK_T format 7-9
system time to readable time 7-6
Cooked mode processing
definition 5-49
CPU
getting time used by process 3-27
scheduling priority, getting 3-27
CPU scheduling priority
getting process, with PROC2_$GET_INFO 3-27
CPU time
getting process, with PROC2_$GET_INFO 3-27
monopolizing with busy-waits 6-3
used by process 3-27
waiting for events without using 6-1
Creating

your own icon font file 5-38
a new process during program execution 3-6
a new window in icon format 5-4
a window in icon format 5-34
backup objects 4-8
icons 5-34
magtape descriptor object, a 9-12
new pads and windows, alternatives to 5-4
new pads and windows, when to 5-4
new transcript pads, when to 5-4
objects 4-5
paste buffers 5-1
temporary objects 4-9
windows and window panes 5-2
CTRL/N 5-10
CTRL/Q 2-25, 6-12, 6-13, 8-19
fault reported, process quit 2-9
CTRL/Y 5-10
CTRL/Z 2-6
Current font size
specifying current height in PAD_$MOVE 5-27
Current scale factors
PAD_$INQ_WINDOWS positions 5-15
Cursor position reports
aligning on word boundaries 5-55
defining a packed record 5-54
Cursors
in raw mode 5-49
reading keyboard cursor position 5-49
vertical moves (frames) 5-3
vertical moves within frames 5-42
Cutting and pasting text and graphics
paste buffers 5-57

Data types
array of records 1-7
arrays 1-7
basic 1-9
declaring DOMAIN predefined data types in FORTRAN 1-1
emulating predefined in C 1-30
enumerated types 1-4
FORTRAN 1-13
how to use 1-4
listed in insert files 1-1
records 1-6
sets 1-5
using DOMAIN predefined 1-1
using reference material 1-9
variant records 1-6
Date
See also time
DEBUG
display with the -SRC option (figure) 5-3
example of frames 5-43, 5-42
example of PAD_$INQ_VIEW 5-43
example of PGM_$DEL_ARG 3-22
example of window panes 5-3
Decoding
definition 8-1
Defining program keys
example of 5-21
Deleting an argument from the argument vector 3-23
Deleting arugments with PGM_$DEL_ARG 3-22
Dereferenced pointer
EC2_$READ 6-6
Determining which file modified recently 7-13
DIR object type
definition 9-2
Directories
C sample program A-64
getting and setting 4-24
no room for objects, error 1-12
object type UID 9-2
DIRECTORY_$UID 4-7
Disabling asynchronous faults with EC2_$WAIT 6-14
Disabling asynchronous faults with EC2_$WAIT_SVC 6-16
Disk files
object type UID 9-2
Display

C sample program A-77
Display driver
See also SMD display driver
Display Manager
bypassing system input processing with raw mode 5-49
changing display appearance with PAD calls 5-1
changing how windows look 5-18
changing scale factors 5-25
closing windows and window panes 5-10
controlling system output with cursors 5-52
controlling user's display with PAD_$INQ_FULL_WINDOW 5-17
creating a new pad in a new window 5-4
creating a new pad in a window pane 5-5
creating and closing windows, sample program 5-11
creating edit window panes 5-9
creating icons 5-34
creating input pads in window panes 5-7
creating new pads and windows 5-4
creating read-only edit pads 5-10
creating subsequent pads in window panes 5-6
creating transcript pads 5-8
displaying text with PAD system calls 5-1
exchanging data with stream records 5-49
formatting window or window pane 5-56
getting current scale factors 5-27
giving user control over the display 5-4
handling graphic input with frames 5-42
inquiring about the user's display and keyboard 5-21
manipulating windows 5-14
overview of 5-2
running programs in user's Shell 5-4
sending and receiving program input 5-49
writing to an output stream 5-56
Display manager commands
copy (XC) 5-57
create process (CP) 5-4
create process only (CPO) 5-4
cut (XD) 5-57
Debug_Quit (DQ) 2-11
font load (FL) 5-23
paste (XP) 5-57
Display unit, value 5-5
DOMAIN Language Level Debugger
See also DEBUG
DOMAIN System Call Reference
data type sections 1-9
error descriptions 1-12
explanation of record illustrations 1-10
explanation of variant record illustrations 1-11
parameter descriptions 1-12
system call descriptions 1-11
using 1-9
DSP server 9-8
Dumb terminal
handling text output as if 5-42
DUP, UNIX function 4-5
EC2 system calls
data types 6-1
insert file 6-1
program examples 6-5
summary of, table 6-1
waiting for multiple events 6-4
See also Eventcount
EC2_$EVENTCOUNT 3-8
EC2_$EVENTCOUNT_T 6-1
EC2_$INIT 3-8
EC2_$PTR_T 6-1
EC2_$READ 6-4
EC2_$READ 3-7, 7-18
array of trigger values 6-6

dereferenced pointer 6–6
example 6–7
invalid use of 6–6
EC2_$WAIT 3–7, 6–4, 7–18
example 6–9
format 6–7
handling asynchronous faults 6–13
responding to events 6–7
returning values of satisfied events 6–7
EC2_$WAIT_QUIT 2–6, 6–16, 6–17
EC2_$WAIT_SVC 2–6, 6–4
example of 6–16
handling asynchronous faults 6–13
preventing faults from occurring 6–17
responding to events 6–7
EDFONT
creating your own icon font file 5–38
font editor 5–23
Edit pads
creating 5–9
creating new permanent files 5–9
definition 5–3
example of 5–9
file access privileges 5–10
formatting program input 5–3
getting user input with PAD_$EDIT_WAIT 5–10
specifying existing pathname and namelength 5–9
EDMTDESC, DOMAIN command 9–13
Efficiency
avoiding duplication of existing programs 3–1
creating a new process 3–6
invoking a program within your own process 3–2
passing invoked programs 3–2
Encoding
definition 8–1
including literal text in directives 8–5
End–of–file 3–12
Enumerated types
DOMAIN predefined data types 1–4
emulating in FORTRAN 1–16
internal representation 1–16
reference material explanation 1–16
EOF 4–4
EQUIVALENCE 1–23
Error messages
error creating file 2–6
file already exists 2–4, 2–6
name not found 2–4
Error stream 4–4
ERROR_$CODE 2–2
ERROR_$FAIL 2–2
ERROR_$INIT_STD_FORMAT 2–4
ERROR_$MODULE 2–2
ERROR_$PRINT 2–3
ERROR_$PRINT_FORMAT 2–4
ERROR_$STD_FORMAT 2–4
ERROR_$SUBSYS 2–2
Errors
accessing status code fields with FORTRAN 2–2
explanation of reference material 1–12
FORTRAN error routines 2–2
insert files 2–1
interpreting the status code 2–2
IOS_$BUFFER_SIZE_TOO_SMALL 4–35
IOS_$BUFFER_SIZE_TOO_SMALL 4–28
IOS_$CONCURRENCY_VIOLATION 4–14
IOS_$END_OF_FILE 4–29
IOS_$PUT_CONDIONAL_FAILED 4–25
NAME, error section 1–12
printing an error message in standard error format 2–5
printing error messages 2–3
setting a severity level 2–7
simple error handling procedure 2–3
standardized error reporting 2–4
status code structure 2–1
testing for 2–3
testing for specific STREAM errors 2–7
See also error messages, faults

Eventcount
alternative to, busy-waiting 6-3
best use for 6-4
C sample program A-11
checking if an event occurred 6-9
common errors 6-6
controlling file access with 6-1
declaring replacement, for terminated process 3-8
definition 6-1
FORTRAN declarations 6-1
handling multiple events 6-4
incrementing the trigger value 6-9
incrementing time 6-9, 7-19
overview of 6-2
preventing interruption during wait cycle 6-14
process eventcounts, how they differ from others 3-8
processing the event 6-9
reading current value of 6-6
relationship between process and (figure) 6-2
returning number of satisfied 6-7
satisfying before wait loop 6-8
specifying, with pointers 6-1
summary of program response after fault occurs 6-18
system constraints 6-4
system-defined 6-1
testing for pre-existing input 6-8
trigger value 6-2, 6-10
unexpected results 6-4
user-defined 6-1
waiting for processes with PGM_$GET_EC 3-7
Eventcounts
C sample program A-108
Exchanging data with stream records
PAD system calls 5-49
EXIT 5-10

Fail bit, status code 2-2
FALSE, Boolean value 1-13
Fault handler
C sample program A-17
Fault handlers
backstop 2-24
default 2-24
determining action to take 2-21
establishing 2-21
establishing function as a 2-22
fault record 2-21
function 2-21
ignoring with PFM_$INHIBIT 6-13
invoking process in background mode 3-12
multi-level 2-24
performing clean-up 6-13
PFM_$CLEANUP 6-13
PFM_$ESTABLISH_FAULT_HANDLER 6-13
PFM_$INHIBIT 6-13
responding to faults 6-13
responding to specified faults 2-23
summary of 6-18
types of 2-24
See also Faults
Fault hanlder
C sample program A-18
Fault record
definition 2-21
FAULT_$ACCESS_VIOLATION 2-9
FAULT_$ADDRESS_ERROR 2-9
FAULT_$BLAST 2-10
FAULT_$BUS_TIMEOUT 2-9
FAULT_$CACHE_PARITY 2-10
FAULT_$CHK_INST 2-9
FAULT_$CONTINUE_PROC 2-10
FAULT_$DISPLAY_QUIT 2-9
FAULT_$ECCC 2-9
FAULT_$ECCU 2-9
FAULT_$FAULT_LOST 2-10
FAULT_$FP_BSUN 2-10
FAULT_$FP_DIV_ZERO 2-10
FAULT_$FP_INEXACT 2-10
FAULT_$FP_OP_ERR 2-10
FAULT_$FP_OVRFLO 2-10
FAULT_$FP_SIG_NAN 2-10
FAULT_$FP_UNDFLO 2-10

FAULT_$ILLEGAL_CORPROC 2-10
FAULT_$ILLEGAL_INST 2-9
FAULT_$ILLEGAL_SVC_CODE 2-9
FAULT_$ILLEGAL_SVC_NAME 2-9
FAULT_$ILLEGAL_USP 2-9
FAULT_$INTERRUPT 2-10
FAULT_$INVALID_STACK 2-10
FAULT_$INVALID_USER_FAULT 2-9
FAULT_$NOT_IMPLEMENTED 2-10
FAULT_$NOT_VALID 2-9
FAULT_$NULLPROC_ONB 2-9
FAULT_$PARITY 2-10
FAULT_$PBU_USER_INT_FAULT 2-9
FAULT_$PRIV_VIOLATION 2-9
FAULT_$PROT_VIOLATION 2-9
FAULT_$QUIT 2-9
FAULT_$SINGLE_STEP 2-9
FAULT_$SPURIOUS_PARITY 2-10
FAULT_$STOP 2-10
FAULT_$SUSPEND_PROC 2-10
FAULT_$SUSPEND_PROC_KBD 2-10
FAULT_$TRAPV_INST 2-9
FAULT_$UNDEFINED_TRAP 2-9
FAULT_$UNIMPLEMENTED_INST 2-9
FAULT_$WCS_PARITY 2-10
FAULT_$WHILE_LOCK_SET 2-10
FAULT_$ZERO_DIVIDE 2-9
Faults
  asynchronous 2-11
  correcting before continuing process 2-21
  definition 2-1
  handling 2-8
  handling, with clean-up handlers 2-12
  handling, with fault handlers 2-21
  inhibiting asynchronous faults 2-25
  insert files 2-1, 2-8
  keeping track of inhibited asynchronous faults 2-25
  listing of (table) 2-8
  responding to specified 2-23
  restarting 2-8
  restoring disk files before exiting program 2-12
  synchronous 2-11
    See also Clean-up handlers, errors, fault handlers, synchronous
File access
  performing sequential 4-29
  random 4-31
    See also random access
File access privileges
  after editing session 5-10
  changing with STREAM_$REDEFINE 5-10
File acess
  controlling with eventcounts 6-1
File already exists 2-4
File attributes
  inquiring about 4-24
  pad 5-2
File attributes, changing
  C sample program A-34
File sequence number
    See also Magnetic tape descriptor object
Files
  accessing magtape 9-12
  C sample program A-37, A-39, A-43
    See also objects
Files, UASC
  C sample program A-41, A-45
Files, updating
  C sample program A-48
Floating point
  faults reported 2-10
Font file
  in directory /SYS/DM/FONTS 5-23
Font files
  bold (.b extension) 5-23
  Changing character fonts with EDFONT 5-23
  definition 5-23
  inverted (.iv extension) 5-23
  italics (.i extension) 5-23
  maximum number in use (100) 5-23
  reverse-video (inverted fonts) 5-23
  standard 5-23

variations of typefaces, fonts, size 5-23
Font size
adjusting window size according to 5-32
Fonts
C sample program A-83, A-96
Format directive
definition 8-4
Formatting
ASCII character strings 8-4
ASCII data with %a 8-5
ASCII data with %a (table) 8-7
floating point data 8-8
integer data with %o, %d, %h format directives 8-10
numbers 8-4
producing spaces, new lines, tabs 8-4
program input with edit pads 5-3
special control string directives 8-12
time 7-7
variables with VFMT 8-1
Formatting variables
See also VFMT
FORTRAN
accessing fields of status code with 2-2
array 1-23
Boolean data type 1-13
carriage control characters 4-16
clearing a bit from a large set 1-21
constructing record-like structures 1-10
declaring DOMAIN predefined data types 1-1
declaring eventcounts 6-1
determining which bits are set 1-19
emulating a Boolean value in a record structure 1-14
emulating a Boolean value with LOGICAL 1-13
emulating large sets 1-21
emulating predefined data types in 1-13
emulating records 1-22
emulating variant records 1-24
enumerated types 1-16
EQUIVALENCE 1-23
equivalences, variant arrays 1-28
error routines 2-2
ICHAR transfer function 1-14
logical unit numbers 4-3
parameter descriptions 1-12
passing integer constants as parameters 1-29
passing parameters to system calls 1-29
pointers 1-14
referencing based variables 1-16
set emulation calls in FTNLIB library 1-21
setting a bit in a large set 1-21
setting bits 1-18
special cases for emulating sets 1-21
stream ID 4-3
testing a bit in a large set 1-21
testing a group of bits 1-20
testing bits 1-19
testing for TRUE and FALSE 1-14
testing return status of system call 2-3
using predefined constants and values 1-4
using predefined sets 1-18
using set emulation calls in FTNLIB library 1-18
See also FORTRAN sample programs
FORTRAN sample programs
declaring a Boolean value with LOGICAL variable 1-13
emulating a record 1-23
emulating a variant record 1-25
emulating status code variant record 1-28
handling pointers 1-15
passing an integer parameter 1-29
setting bits 1-18
testing a group of bits 1-20
testing bits one-by-one 1-19

using enumerated types 1-17
using set emulation calls in FTNLIB 1-21
Frames
C sample program A-96
clearing 5-43
closing 5-43
common error 5-42
controlling output in with cursor 5-52
creating 5-42
definition 5-42
deleting frames 5-43
destroying transcript pad 5-43
filling entire transcript pad with 5-42
handling graphic input with 5-42
making re-draw operation efficient 5-43
maximum size 5-42
restrictions with graphics primitives 5-42
specifying height of 5-42
two-dimensional program input 5-3
using PAD_$INQ_FONT, program example 5-27
using, program example 5-29
FTNLIB library 1-18, 1-34
Functions
copying a string to a buffer 5-44
establishing clean-up handler within 2-12
fault handler operation 2-21
releasing clean-up handler 2-14
setting a severity level 2-7
Get call
definition 4-27
GET_EC system call 6-4
pointers to eventcounts (table) 6-5
returned pointer array 6-5
Get_num_arg
internal procedure 5-30
Getting and reading eventcounts 6-5
Getting and reading system-defined eventcounts 6-6
Getting information about your process 3-27
Getting process information
PGM, PM, PROC1, PROC2 system calls 3-27
Getting system time 7-2
GMF
See also graphic map files
GPR
See also Graphics primitives
GPR_$ENABLE_INPUT 1-21
GPR_$GET_EC 6-5
GPR_$INIT 3-7
GPR_$INQ_VIS_LIST 1-8
GPR_$KEYSET_T 1-21
GPR_$TERMINATE 3-7
GPR_$WINDOW_LIST_T 1-7
Graphic Map files (GMF)
paste buffers 5-58
Graphics
viewport 5-2
Graphics input
synchronizing events with eventcounts 6-1
Graphics primitives
child process restrictions 3-7
inquiring about display attributes 5-21
mixing graphics and text in windows 5-1
more complex graphics than frames 5-42
performing graphic operations with 5-1
restrictions with frames 5-42
without input pads 5-7
Greenwhich Mean Time 7-1
Growing windows
with PAD_$SET_FULL_WINDOW 5-17
Handling
asynchronous faults during eventcount waits 6-13
asynchronous faults with a time eventcount 6-15

asynchronous faults with EC2_$WAIT_SVC 6-17

faults with PFM_$ESTABLISH_FAULT_ HANDLER 6-13

faults, techniques to 6-13

pointers in FORTRAN (example) 1-15

runtime errors 2-1

the touchpad and mouse 5-1

HDR_UNDEF_$UID 4-7

HDRU object type

definition 9-2

Hexadecimal data

converting to ASCII 8-1

Hold_display

internal procedure to suspend process 5-12

How the system represents time 7-1

Hysteresis factor

changing value 5-64

default value (5) 5-64

definition 5-64

I/O

access type and concurrency mode 4-12

changing record formats 4-45

current position marker 4-4

determining operations for types 4-18

device-dependent 4-1

error I/O 4-3

insert files 4-1

interactive 4-4

IOS interface 4-1

overview 4-2

predefined streams 4-3

reading and changing object attributes 4-16

redirecting standard I/O 4-4

referring to stream by number 4-3

seek keys 4-4

standard streams 4-4

writing to an object 4-25

See also accessing DOMAIN objects, IOS Calls

IADDR, FORTRAN function 2-23

ICHAR transfer function 1-14

Icons

C sample program A-89, A-92

changing position of 5-36

creating a new window in icon format 5-4

creating your own 5-35

creating your own icon font 5-38

PAD system calls to manipulate (table) 5-34

positioning 5-35

remembering position of 5-36

replacing icon character 5-36

specifying icon character displayed 5-35

Ignoring asynchronous faults with PFM_$INHIBIT 6-13

Inhibit count, definition 2-25

Input pads

creating, in window panes 5-7

definition 5-2

specifying maximum height 5-7

specifying null pathname and namelength 5-7

specifying PAD_$INIT_RAW 5-8

INPUT_PAD_$UID 4-7

Insert files

BASE 1-1, 1-9, 4-4

CAL(calendar) 7-1

EC2 6-1

ERROR 2-1

FAULT 2-8

how to use 1-1

IOS 4-1

PAD 5-1

PBUFS 5-1

PFM 2-1

PGM 3-1

PM 3-1

predefined constants and values 1-3

PROC1 3-1

PROC2 3-1

summary of (table) 1-2

TIME 7-1
TPAD 5-2
type UID 4-1, 4-6
VFMT 8-1
Integers
formatting 8-4
Interactive I/O 4-4
Interpreting
the status code 2-2
Invoking
a child process, things to consider 3-6
a program in background mode 3-13
an independent process 3-11
clean-up handlers 2-3
multiple clean-up handlers 2-15
other programs within a program 3-1
system-defined programs within a program 3-1
Invoking a program in default mode
creating a new process 3-6
knowing when program is done 3-7
waiting for a child process 3-7
Invoking external programs
coordinating programs through severity levels 3-3, 3-4
in wait mode 3-2
invoking a Shell command 3-3
severity levels 3-3
Invoking programs
PGM_$INVOKE 5-4
setting a severity level 2-7
IOS calls
accessing DOMAIN objects with 9-1
accessing magtape files with 9-12
accessing mailboxes with 9-3
accessing serial lines with 9-8
controlling access type, concurrency 4-12
controlling open call 4-11
creating, opening objects 4-5
implementing object types 4-2
manipulating streams 4-5
performing system I/O 4-1
specifying an object's type 4-6
using with SIO type objects 9-9
See also I/O
IOS_$BUFFER_TOO_SMALL 4-28
IOS_$CF_APPEND 4-17
IOS_$CF_IPC 4-17
IOS_$CF_READ_INTEND_WRITE 4-17
IOS_$CF_TTY 4-17
IOS_$CF_UNREGULATED 4-17
IOS_$CF_VT 4-17
IOS_$CF_WRITE 4-17
IOS_$CLOSE 4-24
IOS_$CONCURRENCY_VIOLATION 4-14
IOS_$COND_OPT 4-25, 4-28, 9-4, 9-5
IOS_$CREATE 4-5
IOS_$DELETE 4-24, 9-1
IOS_$DUP 4-5
IOS_$EQUAL 4-5
IOS_$ERRIN 4-4
IOS_$ERROUT 4-4
IOS_$EXPLICIT_F2 4-45
IOS_$F1 4-45
IOS_$F2 4-45
IOS_$FORCE_WRITE_FILE 4-24
IOS_$GET 4-27
IOS_$GET_DIR 4-24
IOS_$INQ_BYTE_POS 4-31
IOS_$INQ_CONN_FLAGS 4-18
IOS_$INQ_FILE_ATTR 4-24
IOS_$INQ_FULL_KEY 4-32
IOS_$INQ_MGR_FLAGS 4-18
IOS_$INQ_OBJ_FLAGS 4-18
IOS_$INQ_PATH_NAME 4-24
IOS_$INQ_REC_POS 4-31
IOS_$INQ_REC_REMAINDER 4-28
IOS_$INQ_REC_TYPE 4-44
IOS_$INQ_SHORT_KEY 4-32
IOS_$INQ_TYPE_UID 4-24
IOS_$INQUIRE_ONLY_OPT 4-12
IOS_$LOC_NAME_ONLY_MODE 4-8, 4-9
IOS_$LOCATE 4-27
IOS_$MAKE_BACKUP_MODE 4-8
IOS_$MF_CREATE 4-18

IOS_$MF_CREATE_BAK 4-18
IOS_$MF_FORCE_WRITE 4-18
IOS_$MF_FORK 4-18
IOS_$MF_IMEX 4-18
IOS_$MF_READ_INTEND_WRITE 4-18
IOS_$MF_REC_TYPE 4-18
IOS_$MF_SEEK_ABS 4-18
IOS_$MF_SEEK_BOF 4-18
IOS_$MF_SEEK_BYTE 4-18
IOS_$MF_SEEK_FULL 4-18
IOS_$MF_SEEK_REC 4-18
IOS_$MF_SEEK_SHORT 4-18
IOS_$MF_SPARSE 4-18
IOS_$MF_TRUNCATE 4-18
IOS_$MF_UNREGULATED 4-18
IOS_$MF_WRITE 4-18
IOS_$NO_OPEN_DELAY_OPT 4-11
IOS_$NO_PRE_EXIST_MODE 4-8
IOS_$OF_ASCII 4-16
IOS_$OF_COND 4-16
IOS_$OF_DELETE_ON_CLOSE 4-16
IOS_$OF_FTNCC 4-16
IOS_$OF_SPARSE_OK 4-16
IOS_$OPEN 4-5
IOS_$PARTIAL_RECORD_OPT 4-25, 4-38
IOS_$POSITION_TO_EOF_OPT 4-11
IOS_$PRESERVE_MODE 4-8
IOS_$PREVIEW_OPT 4-25, 4-28
IOS_$PUT 4-25
IOS_$PUT_CONDIONAL_FAILED 4-25
IOS_$READ_INTEND_WRITE_OPT 4-12
IOS_$REC_BNDRY_OPT 4-28
IOS_$RECREATE_MODE 4-8
IOS_$REPLICATE 4-5
IOS_$SEEK 4-41, 9-1, 9-18
IOS_$SEEK_TO_BOF 4-31
IOS_$SEEK_TO_EOF 4-31
IOS_$SET_CONN_FLAG 4-18
IOS_$SET_DIR 4-24
IOS_$SET_OBJ_FLAG 4-18
IOS_$SET_REC_TYPE 4-44
IOS_$STDIN 4-4
IOS_$STDOUT 4-4
IOS_$SWITCH 4-5
IOS_$TRUNCATE 4-25, 9-1
IOS_$TRUNCATE_MODE 4-8
IOS_$UNDEF 4-45
IOS_$UNREGULATED_OPT 4-11
IOS_$V1 4-45
IOS_$WRITE_OPT 4-11
IPAD object type
    definition 9-2
IPC sockets
    synchronizing events with eventcounts 6-1
IPC_$GET_EC 6-5

Keyboard
    C sample program A-77
    definitions,redefining 5-21
    type, getting user's 5-21
Keyboard cursor
    definition 5-52

LIB_$ADD_TO_SET 1-21, 1-34
LIB_$CLR_FROM_SET 1-21, 1-34
LIB_$INIT_SET 1-21, 1-34
LIB_$MEMBER_OF_SET 1-21, 1-34
Line, definition 5-3
Local time
    C sample program A-114, A-123, A-125
    computed from UTC 7-2
    getting local time in readable format 7-3
    in TIME_$CLOCK_T format 7-2
    removing local offset 7-4
Locator device
    changing speed of cursor movement 5-62
    corresponding to absolute position on screen 5-61
    jumping cursor 5-62
    making it less sensitive to slight movement (hysteresis factor) 5-64
    making system *forget* last position 5-64

responding to current position only (relative mode) 5–62
responding to initial position and current positions 5–62
scale factors 5–63
stroking 5–62
See also mouse or touchpad

Locked resources
suspending process during 7–16

LOGICAL type 1–13

Machine instructions
fault detection 2–8

Magnetic tape descriptor object
closing 9–13
definition 9–12
file sequence number 9–12, 9–14
object type UID 9–3

Magtapes
accessing files on 9–12
C sample program A–52, A–56
closing a descriptor object 9–13
creating, opening descriptor object 9–12
descriptor object 9–12
reading from 9–18
reading, changing descriptor object 9–13
restrictions 3–6
retrieving volume or object attributes 9–13
stripping newlines from files 9–14
writing to 9–14

Mailbox
synchronizing events with eventcounts 6–1

Mailbox record
definition 9–4

Mailboxes
C sample program A–62
data messages, types of 9–4
getting response from server 9–4
how data is stored 9–4
MBX interface, overview of 9–3
opening 9–4
program example using IOS calls 9–5
server program 9–5
See also MBX interface

Mailboxes, object type UID 9–2

Manager attributes
definition 4–17

Managing programs
accessing arguments from an invoked program 3–20
deleting arguments 3–22
getting information about other processes 3–29
handling process eventcount 3–8
invoking a program in background mode 3–11, 3–13
invoking external user programs 3–1
invoking programs with PGM, PROC, PM system calls 3–1
passing arguments to invoked programs 3–18
passing correct number of arguments using PGM_$DEL_ARG 3–22
passing streams to an invoked program 3–24
process faults detected 2–10
program level 3–3
See also invoking a child process, invoking a program in default mode, invoking external programs, PGM_$INVOKE, Process information

Manipulating tme 7–1

Mapping files 3–8

MBX interface
client, server relationship 9–3
exchanging data 9–3
initializing MBX object 9–3
making connections to an MBX object 9–3
writing client program with IOS calls 9–4
See also mailboxes

MBX object type 9–1
definition 9–2

MBX server program 9–5

MBX_$CHANNEL_SET_T 1–21

MBX_$CREATE_SERVER 9-3
MBX_$GET_EC 6-5
MBX_$GET_REC_CHAN_SET 1-21
MBX_$OPEN 9-3, 9-4
MBX_$PUT_CHR 9-4
MBX_$PUT_REC 9-4
MBX_$UID 4-7
MODULE 3-11
    Pascal fault-handling function 2-22
Mouse
    operates only in relative mode 5-61
    system calls to control (TPAD) 5-1
    See also locator device
Moving windows
    with PAD_$SET_FULL_WINDOW 5-17
MS calls
    implementing UASC type 4-2
MS_$ACC_MODE_T 1-5
MT object type 9-1
    definition 9-3
MT_$UID 4-7, 9-12
MTS system calls
    creating magtape descriptor with 9-12
    See also magtapes
MTS_$ASCII_NL_A 9-14
MTS_$CLOSE_DESC 9-13
MTS_$COPY_DESC 9-13
MTS_$CREATE_DEFAULT_DESC 9-12
MTS_$FILE_SEQUENCE_A 9-14, 9-18
MTS_$GET_ATTR 9-13
MTS_$HANDLE_T 9-12
MTS_$OPEN_DESC 9-12
MTS_$RW_T 9-12
MTS_$SET_ATTR 9-13, 9-18
MTS_$WRITE 9-12
Multiple events
    handling with EC2 calls 6-4

NAME system calls
    error section 1-12
NAME_$BAD_PATHNAME 1-12
NAME_$DIRECTORY_FULL 1-12
NAME_$PNAME_T 1-9, 1-37
NAME_$SET_DIR 1-29, 1-38
Namelength
    creating temporary pad with null value 5-5
    specifying null for input pads 5-7
NEWLINE 4-25, 4-28
    control character 5-3
    writing to output, with %/ 8-13
Node
    getting information about processes on same 3-29
Nonlocal GOTO 2-12
NULL_$UID 4-7

OBJ object type
    definition 9-2
Object
    accessing DOMAIN 9-1
Object attributes
    changing 4-17
    definition 4-16
Object types
    and the IOS interface 4-2
    determining I/O operations for 4-18
    type of files 9-2
OBJECT_FILE_$UID 4-7
Objects
    changing attributes 4-16
    closing and deleting 4-24
    compatible access types and concurrency 4-14
    controlling creation of 4-7
    controlling object access 4-12
    creating 4-5
    creating backups 4-8
    creating new, if name exists 4-7
    getting additional information on 4-24
    guidelines for access, concurrency 4-12
    locking 4-12
    options to control reading 4-28
    options to control writing 4-25
    protected and shared concurrency

4-12
random access 4-31
reading from 4-27
reading sequentially 4-29
record 4-35
sequential access 4-28
sharing, on different nodes 4-13
writing to 4-25
writing to fixed-length records 4-36
writing to variable-length records 4-38
writing, NEWLINE characters 4-25
See also Object attributes, record-oriented objects
Opening a stream to the serial line 6-7
Origin value for TPAD calls
definition 5-61
Orphan process
creating 3-17
definition 3-11
See also Child process
Output cursor, definition 5-52
Overwriting text in frames
using PAD_$CLEAR_FRAME 5-43

PAD object type
definition 9-2
PAD system calls
changing the appearance of the display 5-14
control codes 5-56
data types 5-1
handling graphic input with frames 5-42
inquiring about window positions 5-14
insert files 5-1
requiring size of current font 5-25
specifying a window number 5-14
using icons 5-34
PAD_$800_DISPLAY 5-22
PAD_$ABS_SIZE 5-25
PAD_$BOTTOM 5-6, 5-7
PAD_$BS control code 5-56
PAD_$BW_15P 5-22
PAD_$BW_19L 5-22
PAD_$CLEAR_FRAME 5-33, 5-43
PAD_$CLOSE_FRAME 5-43
PAD_$COLOR_DISPLAY 5-22
PAD_$CPR_ALL 5-54
PAD_$CPR_ENABLE 5-54
locating keyboard cursor 5-52
reading keyboard cursor position in raw mode 5-49
requiring current font size 5-25
See also cursor position reports
PAD_$CR control code 5-56
PAD_$CREATE 5-5, 5-6, 5-13
specifying absolute size with PAD_$ABS_SIZE 5-8
PAD_$CREATE_FRAME 5-25, 5-29
moving cursor vertically by creating frames 5-42
PAD_$CREATE_ICON 5-34, 5-35, 5-36, 5-40
PAD_$CREATE_WINDOW 5-4, 5-5, 5-12
PAD_$DEF_PFK 5-23, 5-56
PAD_$DM_CMD 5-57
PAD_$EDIT 5-9
PAD_$ESCAPE control code 5-56
PAD_$FF control code 5-56
PAD_$ICON_WAIT 5-34, 5-36, 5-38
PAD_$INIT_RAW 5-51
PAD_$INPUT 5-7
PAD_$INQ_DISP_TYPE 5-21
PAD_$INQ_FONT 5-27, 5-28, 5-32
PAD_$INQ_FULL_WINDOW 5-18, 5-33
getting position of window borders with 5-17
PAD_$INQ_ICON 5-34, 5-36
PAD_$INQ_ICON_FONT 5-34, 5-38
PAD_$INQ_POSITION 5-25
PAD_$INQ_WINDOWS 5-15
getting window positions on display 5-14
order of returned positions 5-14
program example 5-16
requiring current font size 5-25
specifying a window number with

5–14
PAD_$LEFT 5–6
PAD_$LOAD_FONT 5–23, 5–24, 5–32
PAD_$LOCATE 5–25, 5–52
PAD_$MAKE_ICON 5–34, 5–37
PAD_$MAKE_INVISIBLE 5–18, 5–20
resctriction 5–17
PAD_$MOVE 5–29, 5–33, 5–51
requiring current font size 5–25
specifying 'Y' factor as height of current font 5–27
PAD_$NEWLINE control code 5–55, 5–56, 5–60
PAD_$NONE 5–22
PAD_$POP_PUSH_WINDOW 5–18, 5–20
PAD_$POSITION_T 5–35
PAD_$RAW 5–50, 5–54
PAD_$READ_EDIT 5–10
PAD_$RIGHT 5–6
PAD_$SELECT_WINDOW 5–18
PAD_$SET_AUTO_CLOSE 5–11, 5–13
when not to use 5–11
PAD_$SET_BORDER 5–18, 5–19
restriction 5–18
PAD_$SET_FULL_WINDOW 5–17, 5–18
implementation restriction 5–17
PAD_$SET_ICON_FONT 5–34, 5–35
PAD_$SET_ICON_POS 5–37
PAD_$SET_ICON_POSITION 5–34
PAD_$SET_SCALE 5–15, 5–25
program example 5–15, 5–27, 5–29, 5–32
PAD_$TAB control code 5–56
PAD_$TOP 5–6
PAD_$TRANSCRIPT 5–5
PAD_$UID 4–7
PAD_$USE_FONT 5–23, 5–25
Pads
closing original pad last 5–10
closing pads automatically 5–10
definition 5–2
object type UID 9–2
types of 5–2
See also PAD system calls
Panes
See also Window panes
Parameters
explanation of reference material 1–12
passing by reference 1–37, 1–38
passing in C 1–37
passing in FORTRAN 1–29
passing integer constants 1–29, 1–39
passing integer parameters in C 1–38
See also passing
Parent process
definition 3–6
keeping track of child process 3–7
passing streams to child 3–6
running out of resources 3–7
Parity
fault reported 2–10
Parsing an input line
with VFMT format directives 8–15
Partial data mailbox messages 9–4
Pascal
CHR function 4–25
fault–handling function 2–22
FIRSTOF function 4–19
formatting data with VFMT 8–1
LASTOF function 4–19
testing return status of system call 2–3
variant records 4–38
writeln 4–2
Passing
a program name to an invoked program 3–18
arguments from a Shell 3–18
arguments to an invoked program 3–18
streams to an invoked program 3–24
Paste buffer manager
using paste buffers 5–57
Paste buffers
C sample program A–105
definition 5–57
Image (graphic map) files 5–58
names 5–57

reading and writing 5-57
sample program 5-58
temporary files 5-58
text (UASC) files 5-58
using 5-57
Pathname
creating a new permanent file 5-8
creating temporary pad with null value 5-5
not unique, error 1-12
not valid, error 1-12
specifying exiting file 5-8
specifying null for input pads 5-7
PBU_$GET_EC 6-5
PBUFS system calls
insert files 5-1
PBUFS_$CREATE 5-57, 5-60
PBUFS_$OPEN 5-57, 5-59
Peripheral devices
accessing 9-8
synchronizing events with eventcounts 6-1
See also serial lines
PFM_$ALL_FAULTS 2-23
PFM_$CLEANUP 2-6, 2-12, 2-17, 6-13
PFM_$CLEANUP_SET 2-6, 2-12
PFM_$CONTINUE_FAULT_HANDLING 2-22
PFM_$ENABLE 6-17
PFM_$ENABLE 2-17, 2-25
reenabling faults 6-13
PFM_$ESTABLISH_FAULT_HANDLER 2-21, 2-22, 2-23, 3-12, 6-13
PFM_$FAULT_REC_T 2-21
PFM_$FG_OPT_SET_T 1-18
PFM_$FH_BACKSTOP 1-31
PFM_$FH_BACKSTP 1-18
PFM_$FH_FUNC_VAL_T 2-21
PFM_$FH_MULTI_LEVEL 1-18, 1-31
PFM_$FH_OPT_SET_T 1-5, 1-31, 2-23
PFM_$FUNC_P_T 2-23
PFM_$INHIBIT 2-25, 6-13, 6-18
PFM_$RESET_CLEANUP 2-16
PFM_$RETURN_TO_FAULTING_CODE 2-22
PFM_$RLS_CLEANUP 2-14
PFM_$SIGNAL 2-15
PGM calls
completion status 3-2
waiting for program to finish before invoking 3-2
See also PGM_$INVOKE
PGM_$ARG 3-18
PGM_$ARGV 3-21
PGM_$ARV 3-18
PGM_$BACK_GROUND 3-11
PGM_$CONNV 3-24
PGM_$DEL_ARG 3-22
PGM_$ERROR 3-3
PGM_$ESTABLISH_FAULT_HANDLER 1-18, 1-31
PGM_$EXIT 2-3, 2-16, 3-3
PGM_$FALSE 3-3
PGM_$GET_ARG 3-20, 5-31
PGM_$GET_ARGS 1-15, 3-20
PGM_$GET_EC 3-7, 6-5
PGM_$GET_PUID 3-29
PGM_$INTERNAL_FATAL 3-3
PGM_$INVALID 3-3
PGM_$INVOKE 2-12, 4-4
accessing argument in UNIV_PTR format 3-21
accessing arguments 3-20
argument vector, argument configuration 3-19
background mode 3-2, 3-11
C sample program A-8, A-9, A-10, A-14, A-24, A-26
default mode 3-2
deleting arguments 3-22
getting information about invoked process 3-29
invoking a Shell command within a program 3-3
invoking another program with 3-1
invoking programs with 5-4
passing arguments to invoked programs 3-18
passing streams to an invoked program

3-24
Returning arguments from child process 3-21
setting severity levels 3-3
standard streams 3-24
wait mode 3-2
See also Managing programs, process execution
PGM_$MAKE_ORPHAN 3-17
PGM_$OK 2-16, 3-3
PGM_$PROC_WAIT 3-2, 3-7
PGM_$SET_SEVERITY 2-8, 2-16, 3-3
PGM_$SIGNAL 2-3
PGM_$TRUE 3-3
PGM_$WAIT 3-2
PGM_$WARNING 3-3
PM_$GET_HOME_TXT 3-27
PM_$GET_SID_TEXT 3-27
Pointers
definition 1-14
dereferencing pointers before passing 1-38
DOMAIN FORTRAN extension syntax 1-14
handling in FORTRAN 1-15
illustration of 1-15
referencing its based variables in FORTRAN 1-16
synchronous fault reported 2-11
using array dimension in FORTRAN 1-15
Popping windows 5-18
Predefined constants
FORTRAN 1-4
using 1-2
Predefined values
using 1-2
Preventing
Display Manager from interpreting a contol code 5-56
faults from occurring with EC2_$WAIT_SVC 6-17
your program from being device-dependent 5-21
your program from being interrupted 2-25
your program from being interrupted during wait cycle 6-14
PROC1_$GET_CPUT 3-27
PROC2_$GET_INFO 3-27, 3-29
PROC2_$LIST 3-30
PROC2_$UID_LIST_T 3-30
PROC_$WAIT 3-6
PROC_$WHO_AM_I 3-27
Procedures
releasing clean-up handlers 2-14
Process
faults detected 2-10
terminating normal with clean-up handler 2-12
Process execution
correcting fault before continuing 2-21
invoking a separate process 3-2
invoking a separate process and getting status when terminated 3-2
invoking a separate program in same process 3-2
suspending process for a specified time 7-18
suspending with TIME_$WAIT 7-16
terminating process after fault 2-12
See also Managing programs, PGM_$INVOKE, suspending a process
Process handle 3-2, 3-7
creating an orphan process 3-17
getting information about child process 3-29
invalid 3-11
Process information
C sample program A-28, A-30
getting amount of CPU time used 3-27
getting CPU scheduling priority 3-27
getting CPU time used by process 3-27
getting home directory 3-27
getting information about an invoked process 3-30
getting information about other

processes 3-29
getting process state 3-27
getting process UID 3-27
getting SID log-in identifier 3-27
getting User Program Counter (UPC) 3-27
getting user stack pointer 3-27
getting User Status Register (USR) 3-27
stack base pointer 3-27
Process manager
See also Managing programs, PGM calls, PGM_$INVOKE
Process state
getting, with PROC2_$GET_INFO 3-27
Process UID
with PROC_$WHO_AM_I 3-27
Processing an event with eventcounts 6-9
Program development
avoiding duplication of existing programs 3-1
error reporting for common subroutines 2-5
finding size and shape of user's window 5-14
giving user more control over display 5-4
handling runtime errors 2-1
inhibiting asynchronous faults sparingly 2-25
inquiring about the user's display and keyboard 5-21
invoking other programs within a program 3-1
knowing whether to scale output to fit in window 5-14
moving cursor origin to a menu window with TPAD_$SET_MODE 5-63
performing concurrent processing 3-1
preventing your program from being device-dependent 5-21
running programs in user's Shell 5-4
suspending a process 7-16
task-oriented units 3-1
See also efficiency, Program units
Program examples
MODULE zero_fault_function 3-16
PROGRAM invoke.pas 3-5
PROGRAM invoke_divide 3-13
PROGRAM pad_cpr_enable 5-52
PROGRAM pad_create_icon 5-39
PROGRAM pad_digclk 5-30
PROGRAM pad_filename 5-44
PROGRAM pad_font 5-24
PROGRAM pad_full_window 5-17
PROGRAM pad_inq_disp_kbd 5-21
PROGRAM pad_inq_font 5-28
PROGRAM pad_inq_window_size 5-15
PROGRAM pad_make_icon 5-36
PROGRAM pad_make_windows 5-11
PROGRAM pad_raw_mode 5-50
PROGRAM pad_scale 5-26
PROGRAM pad_window_show 5-19
PROGRAM parse 8-15
PROGRAM pbufs_paste_buffer 5-58
PROGRAM table 8-14
PROGRAM test_vfmt_example 8-19
Program execution
running a program independently 3-11
Program input
allowing user to edit input 5-49
by lines 5-3
control characters 5-3
converting time input to system time 7-7
converting to program variables, example 8-3
cutting and pasting with paste buffers 5-57
decoding a variable, using VFMT_$READ 8-3
handling in input pads 5-7

handling input (common way) 5-49
handling large amount of input with edit pads 5-9
handling with input pads 5-2
parsing an input line with VFMT (example) 8-15
preprocessing input in input pad 5-49
preventing input from echoing on display 5-8, 5-49
processing in cooked mode 5-49
reading strings using a variety of formats with VFMT (example) 8-19
sending and receiving 5-49
specifying PAD_$INIT_RAW 5-8
translating in program-defined variables 8-1
two-dimensional (frames) 5-3
using incomplete stream records as prompts 5-49
Program level, definition 3-3
Program log
recording dialogue with transcript pads 5-3
Program output
control codes 5-56
converting system time to readable time 7-6
escape sequences 5-56
formatting data for (VFMT) 8-1
formatting output with escape sequences 5-57
handling two-dimensional with frames 5-42
writing a newline to output with %/ 8-13
writing a TAB with %nT 8-13
writing a variable to, using VFMT_$WRITE 8-3
Program response
when asynchronous faults are enabled or disabled 6-13
Program results if fault occurs during wait cycle 6-18
Program units
definition 3-1
DOMAIN system calls 3-1
Protected concurrency mode 4-12
Pushing windows 5-18

Random access
definition 4-28
with seek keys 4-32
See also Seek operations
Raster units
converting window width and height to 5-15
Raw mode
bypassing system input processing with 5-49
C sample program A-102
controlling keyboard cursor with 5-52
definition 5-49
Read access 4-12
Read-intend-write access 4-12
Read-only edit pads
definition 5-3
displaying files that cannot be modified 5-10
for user viewing only 5-3
refers to existing files only 5-10
restriction 5-10
Reading
from magtape files 9-18
from mailboxes 9-4
from objects 4-27
from serial lines 9-9
objects sequentially 4-29
objects, options to control 4-28
records 4-41
See also Seek operations
Reading strings using a variety of formats with VFMT format directives (example) 8-19
REC object type 9-1
definition 9-2
Record formats
fixed-length 4-44
implementing fixed-length 4-45
kinds of 4-44
structure of record with count fields

4-45
structure of record without count fields 4-46
unstructured record 4-46
variable-length 4-45
Record length
C sample program A-32
Record-oriented objects
definition 4-35
IOS calls to operate on 4-35
reading from fixed-length 4-41
record formats 4-44
writing to fixed-length 4-36
writing to variable-length 4-38
See also record formats
Recording program dialogue
transcript pads 5-3
Records
changing from fixed- to variable-length 4-45
DOMAIN predefined data types 1-6
emulating in C 1-35
emulating in FORTRAN 1-22
illustration 1-10
reference material explanation 1-22
See also record-oriented objects, variant records
RECORDS_$UID 4-7
Redrawing display efficiently
PAD_$CLEAR_FRAME 5-43
Reestablishing clean-up handlers
example 2-16
Relative mode for TPAD calls
definition 5-62
getting finer resolution with small scale factors 5-62
timing factors 5-63
Relative seeking
defintion 4-31
Relative time, definition 7-9
Relative value
default size of new window pane 5-6
Releasing a clean-up handler 2-14
Repeat loop, for processing events 6-10
Reporting errors 2-4
Resignaling, definition 2-15
Responding to
asynchronous faults with EC2_$WAIT 6-13
events 6-7, 6-9
user keystrokes in raw mode 5-56
Responding to system-defined eventcounts (example) 6-10
Restartable faults, definition 2-8
Restoring disk files before exiting program 2-12
Returning a severity level from an invoked program 3-5
Returning to program after fault occurred 6-13
Reverse-video characters
See also inverted font file
RIW
See also read-intend-write access
RS-232 protocol standard 9-8, 9-9
Runtime errors
See also errors

Sample programs
accessing a single argument from child program 3-20
accessing an object with seek keys 4-33
accessing arguments with PGM_$GET_ARGS 3-21
adding seconds to current time 7-10
building a character table with VFMT 8-14
changing object attributes 4-21
changing window appearance 5-19
checking compatible access, concurrency 4-15
converting child process to orphan process 3-17
converting system time to readable time 7-7
converting time from ASCII input to system time 7-8
coordinating external programs through severity levels 3-4

creating a UASC object 4-9
creating a window to run a clock 5-29
creating and closing windows and window panes 5-11
creating and writing frames 5-43
deleting arguments before invoking child process 3-23
determining most recent time 7-13
establishing a backstop fault handler 2-24
establishing a clean-up handler 2-13
establishing function as fault handler 2-22
fault handling function 2-22
finding out about user's display and keyboard 5-21
getting current height of font 5-28
getting cursor position reports 5-52
getting information about a child process 3-30
getting information about process 3-27
getting information about windows open to pad 5-15
getting local time from UTC value 7-2
getting local time in readable format 7-3
getting timezone name and offset 7-4
in C A-1
inquiring about object attributes 4-19
invoking a background process 3-11
invoking a Shell command within a program 3-3
invoking child processes with eventcounts 3-8
loading and using fonts 5-24
manipulating full windows 5-17
opening a stream to an SIO line 9-10
opening existing object 4-10
parsing an input line with VFMT 8-15
passing output stream to an invoked program 3-25
passing program name and string to invoked program 3-19
PGM_$GET_ARG 3-21
preventing input from echoing in input pad 5-50
printing an error message in standard error format 2-5
processing information in background mode 3-12
reading sequentially 4-29
reading from a magtape 9-19
reading strings using a variety of formats VFMT 8-19
reenabling asynchronous faults 2-17
reestablishing clean-up handler 2-16
responding to specified faults 2-23
seeking fixed-length records 4-41
setting severity level on a clean-up handler 2-16
subtracting seconds from current time 7-11
suspending a process for a number of seconds 7-16
suspending a process until a specified time 7-17
testing return status of system call 2-3
using a clean-up handler 2-6
using icons 5-36, 5-38
using PAD_$SET_SCALE 5-26
using paste buffers 5-58
using time eventcount to wait for user input 7-18
using TIME_$WAIT 7-16, 7-17
writing fixed-length records 4-36
writing messages to a mailbox 9-5
writing to a magtape 9-15
writing to UASC line by line 4-26
writing variable-length records 4-39
See also C sample programs, FORTRAN sample programs

Scale factors
C sample program A-80
changing in TPAD calls 5-63
column numbers 5-25
default factors for touchpad 5-63

getting current 5-27
restoring default 5-25
Scientific notation
tabulating results using 8-1
Seconds
representing time in 7-9
Seek key
definition 4-4
Seek operations
absolute and relative 4-31
accessing an object with seek keys 4-33
full and short seek keys 4-32
keyed 4-31
nonkeyed 4-31
seeking fixed-length records 4-41
Sequential access 4-29
definition 4-28
Serial lines
accessing 9-8
communicating with another device across 9-8
concurrency control 9-8
copying 9-8
determining current line attributes 9-9
eventcount 6-7
eventcount, example of 6-8
HOST_SYNCH mode attribute 9-10
object type UID 9-3
opening streams to 9-8
performing I/O across 9-9
predefined serial line pathnames 9-8
serial port 9-3
setting attributes 9-8
setting baud rate 9-8
Serial port
definition 9-8
Sets
DOMAIN predefined data types 1-5
emulating in C 1-30
emulating large sets in C 1-34
internal representation 1-18
reference material explanation 1-18
Setting a window aside without closing it
icons 5-34
Setting the bit field 1-30
Severity level
definition 2-8
exiting clean-up handler with 2-15
returning from higher program level 3-3
returning, from an invoked program 3-5
setting, in errors 2-7
Shared concurrency mode 4-12
Shared memory, eventcounts 6-3
Sharing objects
on different nodes 4-13
Shell commands
ABTSEV 2-8
example of PGM_$WAIT 3-2
invoking system-provided programs 3-1
invoking within a program 3-2
SID, login identifier 3-27
SIO access
C sample program A-60
SIO object type 9-1
definition 9-3
SIO_$CONTROL 9-9
SIO_$ERR_ENABLE 1-19, 1-31
SIO_$INQUIRE 1-19, 1-31, 9-9
SIO_$UID 4-7, 9-8
SIO_$VALUE_T 1-24
SMD display driver
black-and-white display driver 5-1
handling a customer-provided locator device 5-61
system calls 5-1
Spaces
skipping between, for output 8-13
Sparse object, definition 4-17
Stack base pointer
getting user's 3-27
Stack pointer
fault reported 2-9
getting user's 3-27
Standard I/O
redirecting 4-4

Standard input 4-3
See also Keyboard, Program input
Standard input eventcount 6-6
Standard output 4-3
See also Program output
Standard output streams
STREAM_$STDOUT, STREAM_$ERROUT 5-5
Status code
accessing fields with FORTRAN 2-2
definition 2-1
explanation of reference material 1-12
information contained in each field 2-2
structure 2-1
testing for specific STREAM errors 2-7
Status.all 2-2, 2-3
STATUS_$OK 1-27, 2-3
STATUS_$T 1-9, 2-1
diagram of 2-2
Stream
marker, definition 4-4
Stream connection
attributes 4-17
changing attributes 4-18
default attributes 4-18
definition 4-3
IOS calls for manipulating 4-5
Stream count, definition 3-24
Stream I/O
repositioning seek key 4-32
Stream ID 4-3
definition 4-3
Stream records
control characters 5-49
definition 5-49
exchanging data with 5-49
using incomplete records as prompts 5-49
STREAM_$CLOSE 5-14
closing pads with 5-10
STREAM_$CREATE 3-11
STREAM_$DELETE 2-3
STREAM_$END_OF_FILE 1-27, 2-6, 2-7
STREAM_$ERROUT 2-4, 5-5
STREAM_$GET 2-6
STREAM_$GET_BUF 5-50
STREAM_$GET_CONDITIONAL
checking if event occurred 6-9
example 6-11
STREAM_$GET_EC 6-5, 6-6
STREAM_$GET_REC 5-50
STREAM_$ID 1-9
STREAM_$INVALID_PATHNAME 3-4
STREAM_$NO_STREAM 3-24
STREAM_$PUT_CHR 5-29
STREAM_$PUT_REC 5-33
STREAM_$REDEFINE 5-10
STREAM_$SEEK 3-12
STREAM_$STDOUT 5-5
STREAM_$SUBS 2-7
STREAM_$UNREGULATED 9-8
Streams
C sample program A-32, A-34, A-37, A-39, A-41, A-43, A-45, A-48, A-52, A-56, A-60, A-62, A-64
copying 4-5
passing a null value as standard input 3-25
passing to an invoked program 3-24
replacing 4-5
standard streams invoked with every program 3-24
synchronizing events with eventcounts 6-1
testing for specific STREAM errors 2-7
Streams facility
definition 4-2
Stringcopy
function copying strings to a buffer 5-44
Subroutines
establishing clean-up handler within 2-12
flexible error reporting 2-5
releasing clean-up handler 2-14
Summary of EC2 system calls, table 6-1

Supplying your own icon characters 5-38
Suspending a process
for a number of seconds 7-16
until a specified time 7-16
with eventcounts 6-2
See also eventcount, time
Synchronizing events with eventcounts 6-1
Synchronizing programs
PGM_$INVOKE 3-2
with shared writing 4-13
Synchronous faults
types of program (table) 2-11
types of system (table) 2-11
System calls
as program units 3-1
CAL (calendar) 7-1
descriptions 1-11
EC2 6-1
ERROR 2-1
error descriptions 1-12
formatting error messages (example) 2-5
how to use insert files 1-1
IOS 4-1
PAD 5-1
parameter descriptions 1-12
PBUFS 5-1, 5-57
PFM 2-1
PGM 3-1
PM 3-1
PROC1 3-1
PROC2 3-1
reference material 1-9, 1-11
returning a character array 1-38
returns system error condition 2-1
specifying input parameters 1-3
system prefixes 1-1
testing return status 2-3
TIME 7-1
TPAD 5-1, 5-61
using DOMAIN predefined 1-1
using predefined constants and values 1-2
VFMT 8-1
why DOMAIN uses predefined data types 1-1
See also data types
System errors, definition 2-1
System-defined eventcounts 6-4

TAB
control character 5-3
writing to output with %nT 8-13
Temporary objects
creating 4-9
Terminating a program 6-13
Terminating normal procesing 2-12
Testing for errors 2-3
Testing for pre-existing input before eventcount 6-8
Time
absolute, definition 7-9
adding seconds to current time 7-10
adding time values 7-9
C sample program A-23, A-114, A-115, A-117, A-118, A-120, A-123, A-125
CAL_$APPLY_LOCAL_OFFSET 7-2
CAL_$GET_LOCAL_TIME 7-2
CAL_$TIMEDATE_REC_T 7-3
calculating timezone offsets 7-4
comparing time values 7-9
comparing times 7-13
Converting from ASCII to CAL_$TIMEDATE_REC_T 7-7
converting from CAL_$TIMEDATE_REC_T to TIME_$CLOCK_T 7-7
converting internal values to readable output 7-2
converting readable time to system time 7-7
converting relative system value to floating point 7-9
converting relative system value to integer 7-9
converting relative time value to system value 7-9
converting system time to readable

time 7-6
converting whole seconds to TIME_$CLOCK_T 7-10
determining most recent 7-13
getting local 7-2
getting relative time value in fractions of a second 7-9
getting relative time value in whole seconds 7-9
getting timezone name 7-4
getting timezone name and offset 7-4
getting timezone offset 7-4
how the system represents 7-1
incrementing system time 7-18
internal representation 7-1
local offset 7-4
manipulating 7-1
obtaining local time 7-1
relative, definition 7-9
representing time in seconds 7-9
subtracting time values 7-9
subtracting time, checking for negative value 7-11
subtracting times 7-11
suspending process execution with TIME_$WAIT 7-16
user-readable 7-2
using a time eventcount 7-18
See also local time
Time eventcount
C sample program A-108, A-111
incrementing eventcount value 6-9
system increments 6-5
TIME_$CLOCK system routine 7-2
TIME_$CLOCK_T 1-9
converting relative time to 7-9
converting whole seconds to 7-10
internal (system) time value 7-1
TIME_$GET_EC 6-5, 7-18
TIME_$RELATIVE 5-12
TIME_$WAIT 5-12, 7-16
Timezone
C sample program A-115
Timezones
getting local offset 7-4
getting U.S. standard timezones 7-4
Touchpad
making it less sensitive to slight movement (hysteresis factor) 5-64
system calls to control (TPAD) 5-1
See also locator device, TPAD system calls
Touchpad manager, using 5-61
TPAD system calls
changing origin in absolute mode 5-63
changing touchpad sensitivity with scale factors 5-63
hysteresis factor 5-64
insert files 5-2
setting origin in relative mode 5-64
timing factors for touchpad or bitpad in relative mode 5-63
TPAD_$SET_CURSOR
changing origin with 5-63
setting origin in relative mode with 5-64
TPAD_$SET_MODE
changing mode of locator device operation 5-62
changing origin in absolute mode 5-63
fine-tuning relative mode resolution with 5-62
making touchpad less sensitive to slight movement 5-64
specifying small scale factors 5-63
Transcript pads
creating a new permanent file 5-8
creating new windows or panes 5-4
creating, in window panes 5-8
definition 5-3
displaying menus with existing files 5-8
size 5-8
specifying existing file 5-8
specifying null pathname and namelength 5-8
temporary 5-5
Trigger value
incrementing 6-9
multiple eventcounts satisfied 6-8

process eventcount 3-8
returned number of satisfied eventcount 6-7
TRUE, Boolean value 1-13
Truncating unnecessary arguments with PGM_$DEL_ARG 3-22
Type manager
implementing object types 9-1
Type managers
defining new types 4-2
implementing I/O operations 4-2
object attributes 4-17
performing I/O operations 4-6
Type UID
default, UID_$NIL 4-7
definition 4-6
inquiring about 4-24
Types of fault handlers 2-24

UASC object type 9-1
definition 9-2
I/O implementation 4-2
retrieving lines 4-25
UASC_$UID 4-7
UID_$NIL 4-7
UID_$T 1-9
UNIV_PTR 1-14
UNIV_PTR format 3-21
accessing an argument in 3-21
typecast to make explicit pointers 3-21
UNIV_PTR types 3-18
Universal Coordinated Time
See also UTC
User Program Counter (UPC)
getting with PROC2_$GET_INFO 3-27
User Status Register (USR)
getting with PROC2_$GET_INFO 3-27
Using a time eventcount 7-18
Using icons
PAD system calls to (table) 5-34
UTC
definition 7-1
getting current value 7-2
Variables
formatting for output 8-1
interpreting user input for program use 8-1
Variant record
status code 2-1
Variant records
DOMAIN predefined data types 1-6
emulating in C 1-36
emulating in FORTRAN 1-24
illustration 1-11
VFMT
building control strings 8-4
control instructions 8-2
decoding example 8-3
decoding, definition 8-1
encoding, definition 8-1
error message output 2-5
examples 8-14
format directive, definition 8-4
formatting instructions 8-2
formatting variables with 8-1
insert files 8-1
simple examples 8-2
syntax 8-2
using dummy values to fill argument list 8-2
See also formatting variables, VFMT control strings, VFMT format directives
VFMT control strings
end-of-string directive 8-4
including literal text for encode operations 8-5
marking the end of a 8-12
repeating control directive sequence 8-5
special format directives 8-12
VFMT format directives
ASCII data (%A) options 8-7
controlling operation of other directives 8-12
floating point data 8-9
formatting ASCII data 8-5

formatting integer data 8-10
in repeat loops 8-13
including a literal percent sign (%) 8-13
marking the end of a control string 8-12
redefining characters specified by E with % 8-12
skipping spaces between output data 8-13
special control string directives 8-12
summary of (table) 8-5
types of 8-5
writing a newline character to output 8-13
writing a TAB character to output 8-13

VFMT_$DECODE 5-31, 8-1
VFMT_$ENCODE 8-1
VFMT_$READ 8-1
VFMT_$READ2 5-13
VFMT_$RS 8-1
VFMT_$WRITE 8-1
VFMT_$WRITE2 5-22
VFMT_$WRITE5 8-2
VFMT_$WS 8-1
VFMT_$WS2 5-25
VFMT_WRITE2 8-3
Viewport, definition 5-2

Waiting
for a child process 3-7
for a specified amount of time 7-18
for events with eventcounts 6-7
for PGM invoked program to finish 3-2
for system-defined eventcounts 6-8
for two eventcounts 6-8

Window panes
associating additional panes 5-6
creating edit 5-9
creating input 5-7
creating new pad in 5-5
creating read-only edit pads 5-10
definition 5-3
determining size of with [ ] 5-6
example of 5-3
giving user more control over display 5-4
keeping size constant with PAD_$ABS_SIZE 5-8
maximum number allowed 5-6
specifying position of 5-6
when to create 5-4

Windows
attributes 5-2
C sample program A-66, A-69, A-71, A-73, A-85, A-96
Changing the appearance of 5-18
changing the most current window viewing the pad 5-14
changing to icons 5-34
closing 5-10
controlling screen display 5-4
creating in icon format 5-34
definition 5-2
getting information about entire display 5-17
getting position of window borders 5-17
getting window positions on display 5-14
growing and moving windows 5-17
legends, borders, getting position of 5-17
making borderless 5-18
making windows disappear 5-18
popping windows to the foreground 5-18
pushing windows to the background 5-18
remembering last position 5-17
setting future position of 5-5
specifying window number 5-14
when to create 5-4

Write access 4-12

Writing
a fault handling function 2-22
a newline to output 8-13
fixed-length records 4-36

out control codes literally 5-56
to an object 4-25
to magtape objects 9-14
to mailboxes 9-4
to serial lines 9-9
variable-length records 4-38
variables to standard output 8-2
See also program output

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Programming with General System Calls Update 1*
Order No.: 008858 Revision: 00 Date of Publication: March, 1986

What type of user are you?

_____ System programmer; language ____________________
_____ Applications programmer; language ____________________
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the DOMAIN system? ____________________

What parts of the manual are especially useful for the job you are doing? ____________________

What additional information would you like the manual to include? ____________________

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.) ____________________

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD