VMS

digital

Guide to VMS Programming Resources

Order Number AA–LA57A–TE

Guide to VMS Programming Resources

Order Number: AA–LA57A–TE

April 1988

This guide contains practical guidelines for using VMS program development tools. Major VMS program development resources are reviewed and examples of using these resources are provided. References for more specific information on each topic are also provided.

Revision/Update Information: This document supersedes the *Guide to Programming on VAX/VMS*, Version 4.4.

Software Version: VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1988 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4522

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	**CANADA**	**INTERNATIONAL**
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.
In New Hampshire, Alaska, and Hawaii call 603-884-6660.
In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).
Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

PREFACE **xix**

CHAPTER 1 OVERVIEW **1–1**

1.1	**VMS TEXT PROCESSORS**	**1–3**
1.1.1	**EDT Editor**	**1–3**
1.1.1.1	Keypad Editing Mode • **1–3**	
1.1.1.2	Line Editing Mode • **1–3**	
1.1.1.3	Nokeypad Editing Mode • **1–4**	
1.1.2	**VAX Text Processing Utility**	**1–4**
1.1.2.1	EVE • **1–5**	
1.1.2.2	EDT Keypad Emulation in EVE • **1–5**	
1.1.2.3	VT100, WPS, and Numeric Keypad Emulation in EVE • **1–5**	
1.2	**VAX COMPILERS, INTERPRETERS, AND THE ASSEMBLER**	**1–5**
1.2.1	**VAX Common Language Environment**	**1–5**
1.2.2	**VAX Ada**	**1–5**
1.2.3	**VAX APL**	**1–6**
1.2.4	**VAX BASIC**	**1–6**
1.2.5	**VAX BLISS-32**	**1–6**
1.2.6	**VAX C**	**1–7**
1.2.7	**VAX COBOL**	**1–7**
1.2.8	**VAX DIBOL**	**1–8**
1.2.9	**VAX FORTRAN**	**1–8**
1.2.10	**VAX LISP**	**1–8**
1.2.11	**VAX MACRO**	**1–9**
1.2.12	**VAX PASCAL**	**1–9**
1.2.13	**VAX PL/I**	**1–10**
1.2.14	**VAX RPG II**	**1–10**
1.2.15	**VAX SCAN**	**1–11**
1.3	**LINKER**	**1–11**
1.3.1	**Using Options Files**	**1–11**
1.3.2	**Using Image Maps**	**1–11**
1.3.3	**Using LIBRARIAN with the Linker**	**1–12**
1.3.4	**Linker Input and Output**	**1–12**
1.3.5	**Linker Command Summary**	**1–13**
1.3.6	**Additional Features**	**1–13**

1.4	**DEBUGGERS**	**1–14**
1.4.1	**Symbolic Debugger**	**1–14**
1.4.2	**Delta/XDelta Utility**	**1–15**
1.5	**PROGRAMMING UTILITIES**	**1–16**
1.5.1	**Command Definition Utility**	**1–16**
1.5.1.1	Defining a New Command • **1–16**	
1.5.1.2	Modifying the Process and System Command Tables • **1–16**	
1.5.1.3	Creating a New Command Table • **1–17**	
1.5.1.4	Parsing the Command String • **1–17**	
1.5.2	**Librarian Utility**	**1–17**
1.5.2.1	Types of Libraries • **1–18**	
1.5.2.2	Linking Your Program with Libraries • **1–18**	
1.5.2.3	Assigning Logical Names to Libraries • **1–18**	
1.5.2.4	Sharing Code Using Text Libraries • **1–18**	
1.5.2.5	Manipulating Libraries Using the LIBRARY Command • **1–19**	
1.5.3	**Message Utility**	**1–19**
1.5.4	**Patch Utility**	**1–20**
1.5.5	**SUMSLP Utility**	**1–20**
1.5.6	**System Dump Analyzer**	**1–21**
1.5.7	**National Character Set Utility**	**1–22**
1.6	**CALLABLE SYSTEM ROUTINES**	**1–22**
1.6.1	**I/O Operations**	**1–23**
1.6.2	**Security Procedures**	**1–23**
1.6.3	**File Management**	**1–23**
1.6.4	**Memory Management**	**1–23**
1.6.5	**Screen Management**	**1–23**
1.6.6	**Math Operations**	**1–24**
1.6.7	**Event Synchronization**	**1–24**
1.6.8	**Calling Utility Routines**	**1–24**
1.6.9	**Run-Time Library (RTL) Routines**	**1–24**
1.6.10	**System Services**	**1–29**
1.6.11	**Utility Routines**	**1–34**
1.6.12	**VMS Record Management Services**	**1–35**
1.6.12.1	Device Support • **1–36**	
1.6.12.2	VMS RMS File Control Blocks • **1–36**	
1.6.12.3	VMS RMS Record Control Blocks • **1–36**	
1.6.12.4	VMS RMS Macros • **1–37**	
1.6.13	**VMS Record Management Services Utilities**	**1–38**
1.6.13.1	ANALYZE/RMS_FILE • **1–38**	
1.6.13.2	CONVERT and CONVERT/RECLAIM • **1–39**	
1.6.13.3	CREATE/FDL and EDIT/FDL • **1–39**	

1.7 SYSTEM PROGRAMMING 1–40

CHAPTER 2 USING PROCESSES 2–1

2.1 CREATING PROCESSES 2–1
2.1.1 Types of Processes 2–1
2.1.2 Modes of Execution 2–1
2.1.3 Creating Spawned Subprocesses 2–2
2.1.3.1 Creating a Spawned Subprocess Using LIB$SPAWN • 2–2
2.1.3.2 Creating a Spawned Subprocess Using SYS$CREPRC • 2–3
2.1.3.3 Creating a Spawned Subprocess Using PPL$CREATE_PROCESS • 2–4
2.1.3.4 Debugging Within a Subprocess • 2–5
2.1.4 Creating Detached Processes 2–7

2.2 MANAGING PROCESSES 2–8
2.2.1 Obtaining Process Information 2–9
2.2.2 Setting Privileges 2–12
2.2.3 Scheduling Processes 2–12
2.2.4 Changing Process Names 2–13
2.2.5 Controlling Process Execution 2–14
2.2.6 Deleting Processes 2–15

CHAPTER 3 COMMUNICATION 3–1

3.1 COMMUNICATING WITHIN A PROCESS 3–1
3.1.1 Local Event Flags 3–2
3.1.2 Logical Names 3–2
3.1.2.1 Using Logical Name Tables • 3–2
3.1.2.2 Access Modes • 3–2
3.1.2.3 Creating and Accessing Logical Names • 3–2
3.1.3 Command Language Interpreter Symbols 3–5
3.1.3.1 When to Use Global Symbols • 3–5
3.1.3.2 When to Use Local Symbols • 3–6
3.1.4 Creating and Using Global Symbols 3–6
3.1.5 Common Blocks 3–6
3.1.5.1 How the Process Common Block Is Created • 3–6
3.1.5.2 Modifying or Deleting Data in the Common Block • 3–6
3.1.5.3 Specifying Other Types of Data • 3–6

3.2 INTERPROCESS COMMUNICATION 3–7
3.2.1 Mailboxes 3–7

3.2.1.1 Creating a Mailbox • **3–8**
3.2.1.2 Temporary and Permanent Mailboxes • **3–8**
3.2.1.3 Reading and Writing Data to a Mailbox • **3–9**
3.2.1.4 Synchronous Mailbox I/O • **3–10**
3.2.1.5 Immediate Mailbox I/O • **3–12**
3.2.1.6 Asynchronous Mailbox I/O • **3–16**

3.3 SYSTEM INFORMATION 3–20
3.3.1 Timer Statistics 3–20
3.3.2 System Time 3–23
3.3.2.1 Absolute Time Format • **3–23**
3.3.2.2 Delta Time Format • **3–23**
3.3.2.3 Current Time • **3–23**
3.3.2.4 Time Manipulation • **3–24**

3.4 INTERSYSTEM COMMUNICATION 3–26
3.4.1 Requesting a Network Connection 3–26
3.4.2 Completing a Network Connection 3–27
3.4.3 Exchanging Messages 3–28
3.4.4 Terminating a Network Connection 3–30

CHAPTER 4 SYNCHRONIZATION 4–1

4.1 SYNCHRONIZING OPERATIONS WITH EVENT FLAGS 4–1
4.1.1 Types of Event Flags 4–1
4.1.2 General Guidelines for Using Event Flags 4–2
4.1.3 Using Local Event Flags 4–3
4.1.4 Using Common Event Flags 4–4
4.1.4.1 Associating a Name with a Common Event Flag Cluster • **4–4**
4.1.4.2 Temporary Common Event Flag Clusters • **4–4**
4.1.4.3 Permanent Common Event Flag Clusters • **4–5**

4.2 USING ASYNCHRONOUS SYSTEM TRAPS 4–7

4.3 SPECIFYING A TIME FOR PROGRAM EXECUTION 4–8
4.3.1 Using Processes for Timing 4–8
4.3.1.1 Specified Time • **4–9**
4.3.1.2 Timed Intervals • **4–10**
4.3.2 Placing Entries in the System Timer Queue 4–11

4.4 SYNCHRONOUS AND ASYNCHRONOUS SYSTEM SERVICES 4–12

4.5	USING THE LOCK MANAGER	4–13
4.5.1	Requesting a Lock	4–14
4.5.2	Requesting a Null Lock	4–15
4.6	USING THE PARALLEL PROCESSING RUN-TIME LIBRARY ROUTINES	4–15
4.6.1	Using Subprocesses	4–16
4.6.2	Using Spin Locks	4–16
4.6.3	Using Semaphores	4–17
4.6.4	Using Barrier Synchronization	4–17
4.7	WRITING APPLICATIONS FOR A VMS MULTIPROCESSING ENVIRONMENT	4–18
4.7.1	Writable Global Sections	4–18
4.7.2	Synchronization Using Process Priority	4–19
4.8	PASSING CONTROL TO ANOTHER IMAGE	4–19
4.8.1	Invoking a Command Image	4–19
4.8.2	Invoking a Noncommand Image	4–20

CHAPTER 5 SHAREABLE RESOURCES 5–1

5.1	SHARING PROGRAM CODE	5–1
5.1.1	Object Libraries	5–1
5.1.1.1	System- and User-Defined Default Object Libraries • 5–2	
5.1.1.2	How the Linker Searches Libraries • 5–2	
5.1.1.3	Creating an Object Library • 5–2	
5.1.1.4	Managing an Object Library • 5–2	
5.1.2	Text and Macro Libraries	5–3
5.2	SHAREABLE IMAGES	5–3
5.2.1	Transfer Vectors	5–3
5.2.1.1	Why Use Transfer Vectors? • 5–4	
5.2.1.2	Deleting Transfer Vectors • 5–4	
5.2.2	GSMATCH Option	5–5
5.2.3	UNIVERSAL Option	5–5
5.2.4	Creating Shareable Images	5–6
5.2.5	Shareable Image Libraries	5–8
5.2.5.1	Adding or Replacing Shareable Images • 5–8	
5.2.5.2	Listing or Deleting Shareable Images • 5–8	
5.2.6	Linking Shareable Images	5–8
5.2.6.1	Default File Type and Location of Shareable Images • 5–9	

5.2.6.2	Alternate Location of Shareable Images • **5–9**	
5.2.7	**Shared Images**	**5–10**
5.2.7.1	Creating a Shared Image • **5–10**	
5.2.7.2	If the Shared Image Is in Memory • **5–10**	
5.2.7.3	If the Shared Image Is Not in Memory • **5–10**	
5.3	**SYMBOLS**	**5–10**
5.3.1	**Defining Symbols**	**5–11**
5.3.2	**Local and Global Symbols**	**5–11**
5.3.3	**Resolving Global Symbols**	**5–11**
5.3.3.1	Explicitly Named Modules and Libraries • **5–12**	
5.3.3.2	System Default Libraries • **5–12**	
5.3.3.3	User Default Libraries • **5–12**	
5.3.3.4	Making a Library Available for System-wide Use • **5–12**	
5.3.3.5	Macro Libraries • **5–13**	
5.3.4	**Sharing Data**	**5–13**
5.3.4.1	Installed Common Blocks • **5–13**	
5.3.4.2	Global Sections • **5–15**	
5.3.4.3	VMS RMS Shared Files • **5–19**	

CHAPTER 6 SECURITY FEATURES **6–1**

6.1	**RIGHTS DATABASE**	**6–1**
6.2	**SYSTEM SERVICES AND SECURITY**	**6–1**
6.3	**PRIVILEGED IMAGES**	**6–2**

CHAPTER 7 INPUT/OUTPUT OPERATIONS **7–1**

7.1	**CHOOSING I/O TECHNIQUES**	**7–1**
7.1.1	**Simple User I/O**	**7–1**
7.1.2	**Complex User I/O**	**7–2**
7.1.3	**Reading and Writing Data to Files**	**7–2**
7.1.4	**Reading and Writing Data to Devices**	**7–2**
7.1.5	**Broadcast Messages and Special I/O Actions**	**7–2**
7.2	**USING SYS$INPUT AND SYS$OUTPUT**	**7–2**
7.2.1	**Default Input and Output Devices**	**7–2**

7.2.2 Reading and Writing to Alternate Devices and External Files 7–3

7.3 WORKING WITH SIMPLE USER I/O 7–3
7.3.1 Default Devices for Simple I/O 7–3
7.3.2 Getting a Line of Input 7–4
7.3.3 Getting Several Lines of Input 7–5
7.3.4 Writing Simple Output 7–6

7.4 WORKING WITH COMPLEX USER I/O 7–7
7.4.1 Pasteboards 7–8
7.4.1.1 Creating a Pasteboard • 7–9
7.4.1.2 Deleting a Pasteboard • 7–9
7.4.1.3 Setting Screen Dimensions and Background Color • 7–9
7.4.2 Virtual Displays 7–10
7.4.2.1 Creating a Virtual Display • 7–10
7.4.2.2 Pasting Virtual Displays • 7–11
7.4.2.3 Rearranging Virtual Displays • 7–13
7.4.2.4 Removing Virtual Displays • 7–14
7.4.2.5 Modifying a Virtual Display • 7–15
7.4.2.6 Using Spawned Subprocesses • 7–16
7.4.3 Viewports 7–17
7.4.4 Writing Text to Virtual Display 7–17
7.4.4.1 Positioning the Cursor • 7–17
7.4.4.2 Writing Data Character by Character • 7–18
7.4.4.3 Writing Data Line by Line • 7–19
7.4.4.4 Drawing Lines • 7–20
7.4.4.5 Deleting Text • 7–21
7.4.5 Using Menus 7–22
7.4.6 Reading Data 7–23
7.4.6.1 Reading from a Display • 7–23
7.4.6.2 Reading from a Virtual Keyboard • 7–24
7.4.6.3 Reading from the Keypad • 7–25
7.4.6.4 Reading Composed Input • 7–28
7.4.7 Controlling Screen Updates 7–31
7.4.8 Modularity 7–31

7.5 SPECIAL INPUT/OUPUT ACTIONS 7–33
7.5.1 CTRL/C and CTRL/Y Interrupts 7–33
7.5.2 Unsolicited Input 7–36
7.5.3 Type-Ahead Buffer 7–39
7.5.4 Echo 7–40
7.5.5 Timeout 7–41
7.5.6 Lowercase to Uppercase Conversion 7–42
7.5.7 Line Editing and Control Actions 7–42

7.5.8 **Broadcasts** 7–43
7.5.8.1 Default Handling of Broadcasts • **7–43**
7.5.8.2 How to Create Alternate Broadcast Handlers • **7–44**

7.6 **SYS$QIO AND SYS$QIOW SYSTEM SERVICES** 7–45
7.6.1 **Read Operations** 7–46
7.6.2 **Write Operations** 7–49
7.6.3 **Checking the Device Type** 7–50
7.6.4 **Terminal Characteristics** 7–51
7.6.5 **Record Terminators** 7–53
7.6.6 **File Terminators** 7–54

CHAPTER 8 FILE I/O 8–1

8.1 **FILE ATTRIBUTES** 8–1

8.2 **FILE ACCESS STRATEGIES** 8–1
8.2.1 **Complete Access** 8–1
8.2.2 **Record-by-Record Access** 8–1
8.2.3 **Discrete Records** 8–2
8.2.4 **Sequential and Indexed Files** 8–2
8.2.5 **Protection and Access** 8–2
8.2.5.1 Read Only Access • **8–2**
8.2.5.2 Shared Access • **8–2**
8.2.6 **Specifying File Attributes** 8–3

8.3 **LOADING AND UNLOADING A DATABASE** 8–4
8.3.1 **Using SYS$CRMPSC** 8–4
8.3.1.1 Mapping a File • **8–5**
8.3.1.2 User-Open Routine • **8–8**
8.3.1.3 Initializing a Mapped Database • **8–9**
8.3.1.4 Saving a Mapped File • **8–9**
8.3.1.5 Example of Per-Record Processing of Entire Database • **8–10**

8.4 **SORTING AND MERGING SEQUENTIAL FILES** 8–13
8.4.1 **Using the File and Record Interface** 8–14
8.4.2 **Multiple Sort Operations** 8–14
8.4.3 **Passing Key Information** 8–14
8.4.4 **Sorting with the File Interface** 8–15
8.4.5 **Sorting with the Record Interface** 8–16
8.4.6 **Merging with the File Interface** 8–19
8.4.7 **Merging with the Record Interface** 8–21

8.5	**DATA COMPRESSION AND EXPANSION**	**8–25**
8.5.1	**Compression Routines**	**8–26**
8.5.2	**Expansion Routines**	**8–32**
8.6	**LIBRARIAN UTILITY ROUTINES**	**8–35**
8.6.1	**Creating, Opening, and Closing Libraries**	**8–36**
8.6.2	**Adding Modules**	**8–40**
8.6.3	**Deleting Modules**	**8–42**
8.6.4	**Extracting Modules**	**8–43**
8.6.5	**Using Multiple Keys and Multiple Indexes**	**8–45**
8.6.6	**Accessing Module Headers**	**8–48**
8.6.7	**Reading Library Headers**	**8–50**
8.6.8	**Displaying Help Text**	**8–52**
8.6.9	**Listing and Processing Index Entries**	**8–53**
8.7	**FILE DEFINITION LANGUAGE**	**8–54**
8.7.1	**Creating an FDL File**	**8–55**
8.7.1.1	Using the FDL Editor • **8–55**	
8.7.1.2	Using the Characteristics of an Existing Data File • **8–55**	
8.7.2	**Applying an FDL File to a Data File**	**8–57**
8.7.2.1	Creating a New Data File • **8–57**	
8.7.2.2	Modifying an Existing Data File • **8–58**	
8.8	**USER-OPEN ROUTINES**	**8–58**
8.8.1	**Opening a File**	**8–59**
8.8.1.1	Specifying USEROPEN • **8–59**	
8.8.1.2	Writing the User-Open Routine • **8–59**	
8.8.1.3	Setting FAB and RAB Fields • **8–60**	

CHAPTER 9 CONDITION HANDLING **9–1**

9.1	**GENERAL ERROR HANDLING**	**9–1**
9.1.1	**Condition Code and Message**	**9–1**
9.1.2	**Return Status Convention**	**9–2**
9.1.2.1	Testing Returned Condition Codes • **9–3**	
9.1.2.2	Testing SS$_NOPRIV and SS$_EXQUOTA • **9–3**	
9.1.3	**Signaling Mechanism**	**9–5**
9.1.3.1	Default Condition Handling • **9–5**	
9.1.3.2	Changing a Signal to a Return Status • **9–6**	
9.2	**DEFINING CONDITION CODES AND MESSAGES**	**9–7**
9.2.1	**Creating the Message Source File**	**9–7**

9.2.1.1 Specifying the Facility • 9–8
9.2.1.2 Specifying the Severity • 9–8
9.2.1.3 Specifying Condition Names and Messages • 9–9
9.2.1.4 Specifying Variables in the Message Text • 9–9
9.2.2 Compiling and Linking the Messages 9–9
9.2.2.1 Linking the Message Object Module • 9–9
9.2.2.2 Accessing the Message Object Module from Multiple Programs • 9–10
9.2.2.3 Modifying a Message Source Module • 9–10
9.2.2.4 Accessing Modified Messages Without Relinking • 9–10
9.2.3 Signaling User-Defined Codes and Messages 9–10
9.2.3.1 Signaling with Global Symbols • 9–11
9.2.3.2 Signaling with Local Symbols • 9–11
9.2.3.3 Specifying FAO Parameters • 9–12

9.3 CONDITION HANDLERS 9–12
9.3.1 Establishing a Condition Handler 9–14
9.3.2 Writing a Condition Handler 9–14
9.3.2.1 The Signal Array • 9–14
9.3.2.2 The Mechanism Array • 9–15
9.3.2.3 Comparing the Signaled Condition with an Expected Condition • 9–16
9.3.2.4 Exiting From the Condition Handler • 9–17
9.3.2.5 Returning Control to the Program • 9–18
9.3.3 Debugging 9–20
9.3.4 Condition Handler Functions 9–20
9.3.4.1 Modifying Condition Codes • 9–20
9.3.4.2 Displaying Messages • 9–22
9.3.4.3 Chaining Messages • 9–23
9.3.4.4 Logging Messages • 9–24
9.3.5 System-Defined Arithmetic Condition Handlers 9–26

9.4 EXIT HANDLERS 9–26
9.4.1 Establishing an Exit Handler 9–27
9.4.2 Writing an Exit Handler 9–29
9.4.3 Debugging an Exit Handler 9–30

CHAPTER 10 MEMORY MANAGEMENT 10–1

10.1 USING RTL ROUTINES 10–1

10.2 USING SYSTEM SEVICES 10–2
10.2.1 Working with Address Space 10–3
10.2.2 Adjusting Working Sets 10–3

INDEX

EXAMPLES

1–1	Defining a New Command	1–17
1–2	Message Source File	1–19
2–1	Obtaining the Process Name	2–10
2–2	Obtaining Different Types of Process Information	2–11
3–1	Creating a Spawned Subprocess	3–3
3–2	Opening a Mailbox	3–10
3–3	Synchronous I/O Using a Mailbox	3–11
3–4	Immediate I/O Using a Mailbox	3–13
3–5	Asynchronous I/O Using a Mailbox	3–17
3–6	Displaying and Writing Timer Statistics	3–21
3–7	Calculating and Displaying the Time	3–25
3–8	Exchanging Messages	3–28
4–1	Executing a Program Using Delta Time	4–10
4–2	Executing a Program at Timed Intervals	4–11
4–3	Requesting a Null Lock	4–15
5–1	Interprocess Communication Using Global Sections	5–16
7–1	Reading a Line of Data	7–4
7–2	Reading a Varying Number of Input Records	7–5
7–3	Associating a Pasteboard with a Terminal	7–8
7–4	Modifying the Screen Dimensions and Background Color	7–9
7–5	Defining and Pasting Virtual Displays	7–11
7–6	Scrolling Forward Through a Display	7–19
7–7	Scrolling Backward Through the Display	7–20
7–8	Creating a Statistics Display	7–21
7–9	Reading Data from a Virtual Keyboard	7–24
7–10	Reading Data from the Keypad	7–26
7–11	Redefining Keys	7–29
7–12	Using Interrupts to Perform I/O	7–35
7–13	Receiving Unsolicited Input from a Virtual Keyboard	7–37
7–14	Trapping Broadcast Messages	7–44
7–15	Reading Data from the Terminal Synchronously	7–46
7–16	Reading Data from the Terminal Asynchronously	7–48
7–17	Writing Character Data to a Terminal	7–49
7–18	Using SYS$GETDVIW to Verify the Device Name	7–50
7–19	Disabling the HOSTSYNCH Terminal Characteristic	7–52

8–1 Mapping a Data File to the Common Block 8–6
8–2 Using a User-Open Routine 8–8
8–3 Closing a Mapped File 8–10
8–4 Creating a Sequential File of Fixed-Length Records 8–11
8–5 Updating a Sequential File 8–12
8–6 Sorting a Sequential File Using the File Interface 8–15
8–7 Sorting a Sequential File Using the Record Interface 8–17
8–8 Merging Sequential Files Using the File Interface 8–20
8–9 Merging Sequential Files Using the Record Interface 8–22
8–10 Compressing Data 8–27
8–11 Expanding Data 8–33
8–12 Creating, Opening, and Closing a Text Library 8–37
8–13 Adding Modules to a Text Library 8–40
8–14 Deleting Modules from a Text Library 8–42
8–15 Extracting Modules from a Text Library 8–44
8–16 Associating Keys with Modules 8–45
8–17 Listing Keys Associated with a Module 8–47
8–18 Displaying the Module Header 8–49
8–19 Reading Library Headers 8–51
8–20 Displaying Text from a Help Library 8–52
8–21 Displaying Index Entries 8–54
8–22 Creating an FDL File 8–56

FIGURES

5–1 How the Linker Uses Transfer Vector Address 5–4
7–1 Defining and Pasting Virtual Displays 7–12
7–2 Moving a Virtual Display 7–13
7–3 Repasting a Virtual Display 7–14
7–4 Popping a Virtual Display 7–15
7–5 Statistics Display 7–21
9–1 Structure of a Condition Code 9–2
9–2 Structure of a Signal Array 9–15
9–3 Structure of a Mechanism Array 9–16
9–4 Structure of an Exit Handler 9–28

TABLES

1–1	Types of Libraries	1–18
1–2	RTL General Programming Tasks (LIB$) Routines	1–25
1–3	RTL Screen Management (SMG$) Routines	1–27
1–4	Summary of System Services	1–30
1–5	Utility Routine Summary	1–35
1–6	User Control Blocks	1–37
2–1	Detached Processes and Spawned Subprocesses	2–2
2–2	Comparison of LIB$SPAWN, SYS$CREPRC, and PPL$CREATE_PROCESS	2–5
2–3	Routines and Commands for Managing Processes	2–8
2–4	Comparison of Hibernation and Suspension	2–14
4–1	Event Flags	4–2
4–2	Event Flag Routines	4–3
4–3	Time Statistics System Services	4–9
4–4	Lock Manager Routines	4–14
6–1	Security System Services	6–2
7–1	SYS$INPUT and SYS$OUTPUT Values	7–2
7–2	Setting Video Attributes	7–16
9–1	Privilege Errors	9–3
9–2	Quota Errors	9–4

Preface

Intended Audience

This document is intended for experienced programmers working in the VMS operating system environment.

Document Structure

The *Guide to VMS Programming Resources* is designed to help programmers understand and use the features offered by the VMS operating system. This guide is not intended to be a complete description of any one programming language (see the Associated Documents section for related documentation); instead, it focuses on the tasks that typically confront programmers and suggests ways to use the VMS operating system features to accomplish those tasks.

An overview of all programming resources is provided in the first chapter. The rest of the document is organized according to the following programming tasks:

- Using processes
- Communicating with the system, with other programs, and with other program components
- Synchronizing program execution
- Sharing program code and data
- Using system security features
- Completing I/O tasks
- Condition handling
- Allocating and deallocating memory

Associated Documents

For additional information on topics covered in this document, refer to the following documents:

- The documentation set for your programming language
- *VMS Debugger Manual*
- *VMS Command Definition Utility Manual*
- *VMS Librarian Utility Manual*
- *VMS Linker Utility Manual*
- *VMS Message Utility Manual*
- *VMS Patch Utility Manual*
- *VMS SUMSLP Utility Manual*
- *VMS Utility Routines Manual*

- *VMS System Services Volume*
- *VMS Run-Time Library Routines Volume*
- *VMS Record Management Services Manual*
- *VMS Analyze/RMS_File Utility Manual*
- *VMS Convert and Convert/Reclaim Utility Manual*
- *VMS File Definition Language Facility Manual*
- *VMS National Character Set Utility Manual*
- *VMS I/O User's Reference Volume*
- *VMS Delta/XDelta Utility Manual*
- *VMS System Dump Analyzer Utility Manual*
- *VMS Device Support Manual*

Conventions

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.

Convention	Meaning
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

1 Overview

The VMS operating system offers a wide range of tools and resources to make program development efficient for programmers at all levels. Depending on how you program, what language you are using, or the type of program you are developing, you can utilize several VMS resources to make programming easier.

This chapter outlines the VMS tools and resources that are available and explains the capabilities and functions of each and how you might use them. It also lists sources of additional information about each of the topics reviewed.

The following VMS operating system software components and optional software products are described:

- The EDT editor and the VAX Text Processing Utility for creating and editing source files
- Several VAX programming language compilers and interpreters, plus the assembler, including:
  - VAX Ada
  - VAX APL
  - VAX BASIC
  - VAX BLISS-32
  - VAX C
  - VAX COBOL
  - VAX DIBOL
  - VAX FORTRAN
  - VAX LISP
  - VAX MACRO
  - VAX PASCAL
  - VAX PL/I
  - VAX RPG II
  - VAX SCAN
- Programming utilities that perform linking and debugging as follows:
  - Linker Utility
  - VMS Symbolic Debugger and the Delta/XDelta Utility for debugging programs
- Additional programming utilities for program development as follows:
  - Command Definition Utility for creating new DCL-level commands to invoke your program.

- Librarian Utility for creating and managing libraries of modules
- Message Utility for creating new messages for use in your program
- Patch Utility for changing executable code after it has been compiled or assembled
- SUMSLP Utility, a batch-oriented editor, for applying multiple update files to a single input file
- National Character Set Utility for defining collating sequences and string conversion functions
- System Dump Analyzer for determining the cause of a system failure

- Callable system routines that you can call from any high-level or assembly language program are as follows:
  - Run-time library routines
  - System services
  - Utility routines
  - VMS Record Management Services
- The following VMS RMS utilities, which allow you to create different file formats:
  - Analyze/RMS_File Utility
  - Convert and Convert/Reclaim Utilities
  - File Definition Language Utility
- System programming information for I/O operations and device drivers
- VMS operating system features to use for maximizing program efficiency, including:
  - Creating, controlling, and deleting processes
  - Communicating with other components
  - Synchronizing events
  - Sharing resources
  - Using system security
  - Implementing input/output procedures
  - Condition handling
  - Managing memory

1.1 VMS Text Processors

You can use either the EDT editor or the VAX Text Processing Utility (VAXTPU) to create and modify source files.

Choose the text processor that is best for you, based on what you are used to (that is, using a keypad or line commands) and the type of features you want. The Extensible VAX Editor (EVE) is a VAXTPU-based text editing interface. In general, the EVE interface offers more functionality than EDT, especially for complicated editing tasks. With the EVE interface, more line commands and more built-in procedures are available, and you can create multiple windows to view separate files (or buffers) at the same time.

1.1.1 EDT Editor

EDT is an interactive text editor having the following capabilities:

- Three types of editing modes: keypad mode for screen-oriented editing, line mode for line-number editing, and nokeypad mode for defining your own key sequences. You can use any mode you prefer and you can switch back and forth during a single editing session.
- Journaling to protect your editing session in the event of a system interruption.
- Multiple buffers.
- Access to as many files as you need.
- Startup command files to initialize the EDT editing environment to your own needs.
- EDT macros to automate repetitive editing procedures.

For more information on using EDT, refer to the *VAX EDT Reference Manual.*

1.1.1.1 Keypad Editing Mode

In keypad mode, the text is displayed by screen. You use one-key or two-key sequences from the numeric keypad to execute common editing procedures such as cursor movement, deletion, insertion, text search, text replacement, and cut and paste. With keypad mode, you can define your own key sequences in a EDT initialization file to complete individualized editing functions. To define key sequences, you bind a series of nokeypad commands to a particular key sequence not currently being used.

1.1.1.2 Line Editing Mode

In line mode, the text is displayed one line at a time. With this mode, you can access text according to line number. To edit the text, you enter line commands that complete all the editing functions available in keypad mode. However, you cannot define your own key sequences.

1.1.1.3 Nokeypad Editing Mode

Nokeypad mode is used primarily on VT100-series and VT52 terminals. Text appears on the upper lines of the screen. As you type commands, they are displayed at the bottom of the screen. When you press RETURN, EDT processes the commands.

Use the nokeypad commands to create key sequences for keypad mode. You can bind any key sequence to a series of editing commands.

1.1.2 VAX Text Processing Utility

The VAX Text Processing Utility (VAXTPU) is a high-performance, text processor that can be used to create text editing interfaces such as EVE. VAXTPU has the following features:

- A high-level procedure language with several data types, relational operators, error interception, looping, case language statements, and built-in procedures
- A compiler for the VAXTPU procedure language
- An interpreter for the VAXTPU procedure language
- The EVE editing interface which, in addition to the EVE keypad, provides EDT, VT100, WPS, and numeric keypad emulation

With these tools, you can further customize the EVE editing interface or create your own editing interface designed for your own programming needs.

Special features offered with VAXTPU include the following:

- Multiple buffers
- Multiple windows
- Multiple subprocesses
- Text processing in batch mode
- Insert or overstrike text entry
- Free or bound cursor motion
- Learn sequences
- Pattern matching
- Key definition

For most uses, the EVE editing interface is preferable over EDT because of these advanced features and the ability to design your own customized interface within EVE or using VAXTPU. For further information about using VAXTPU, refer to the *Guide to VMS Text Processing* and the *VAX Text Processing Utility Manual.*

1.1.2.1 EVE

The EVE editing interface is installed with VAXTPU. EVE is easy to learn and fast to use. Most common editing functions are accessed by pressing a single key on the EVE keypad. EVE commands and special VAXTPU features and advanced functions are invoked by entering commands on the EVE command line. With EVE, you can design your own editing keypad and learn sequences by using initialization files and section files.

1.1.2.2 EDT Keypad Emulation in EVE

If you are an experienced EDT user, you can redefine the default EVE keypad bindings to emulate an EDT keypad using the EVE command SET KEYPAD EDT. EDT keypad emulation in EVE provides most of the functions of the EDT keypad and binds these functions to the same keys that EDT uses. A subset of EDT line commands, as well as the VAXTPU commands, is also available.

1.1.2.3 VT100, WPS, and Numeric Keypad Emulation in EVE

The EVE editing interface also supports VT100, WPS, and numeric keypad emulation.

1.2 VAX Compilers, Interpreters, and the Assembler

The VMS operating system supports a variety of language compilers and interpreters that translate source code to object code. Each language has features suitable to different types of programming uses. The VAX programming languages are optional software products.

Most of the VAX programming languages can fully utilize the resources of the VMS operating system. All VAX languages can access any of the callable routines (system services, utility routines, run-time library routines, and record management services). Most VAX languages are fully supported by the Symbolic Debugger. VAX APL, VAX DIBOL, and VAX LISP have their own symbolic debugger utility.

1.2.1 VAX Common Language Environment

All VAX languages support mixed-language programming. VAX languages adhere to the VAX Procedure Calling and Condition Handling Standard, that is, a program written in any VAX programming language can contain calls to procedures written in other VAX languages. For more information about the VAX Procedure Calling and Condition Handling Standard, refer to *Introduction to VMS System Routines.*

1.2.2 VAX Ada

VAX Ada for the VMS operating system is a complete implementation of the Ada programming language; it conforms fully to the ANSI standard and is validated by the Ada Validation Office. VAX Ada features include the following:

- The VAX Ada library manager allowing shared use of a compilation library, shared compiled VAX Ada code either by reference or copy, use of individual libraries as sublibraries of team libraries, and automatic recompilation of obsolete units.

- Individual units (subprograms, tasks, packages, generic units) that can be compiled separately.
- Strong typing to ensure the integrity of data types. Type checking is done at compile time.
- Data abstraction to free your programmer from needing to know specifically how VAX Ada implements data types, executable statements, and so forth.
- Ability to define system features (for example, memory size) that can limit program scope for each application.
- Use of tasks within the language to support parallel processing.
- Ada-defined exception handling to recover from error conditions. User-defined exception handling is also available.

1.2.3 VAX APL

The VAX APL interpreter provides a built-in editor, debugger, system communications facility, and file system. It automatically reserves space for variables, formats input and output statements, and manipulates rows and columns of data without loops. It can call another VAX APL program and have data returned as a result.

1.2.4 VAX BASIC

VAX BASIC can be used as either an interpreter or a compiler. Using it as an interpreter, you can execute unnumbered statements at any time. You can also compile source files to create object modules. VAX BASIC is fully supported by the VMS Symbolic Debugger; it can access callable system routines and call procedures written in other languages. All modules are written in position-independent code and can be run as fully reentrant code.

1.2.5 VAX BLISS-32

VAX BLISS-32 supports development of modular software according to structured programming concepts by providing an advanced set of language features. It provides access to most of the hardware features of the VAX system.

VAX BLISS-32 programs include the following features that allow programs to be transported to other DIGITAL computer systems:

- High-level language constructs may be transferred from one machine to another with little or no alteration.
- Machine-specific functions can be separated from the common, mainline code via modularization, macros, and special Library and Require files (separate files that can be invoked from a BLISS program).
- Machine-specific characters can be passed to BLISS data structures with the use of parameters.

1.2.6 VAX C

VAX C is fully supported within the VMS operating system environment; it can use any of the utilities and can invoke the callable system routines. It is a full implementation of the C programming language with the following additional features to improve its performance within the VMS operating system environment:

- Set of structured control flow operators
- Set of mathematical and logical operators
- Data typing and conversions
- Consistent data declarations and data references
- Compiler optimized code, along with listing and cross-referenced storage map
- Common set of run-time support routines for accomplishing common tasks such as I/O or math routines (many UNIX-specific routines have been emulated)
- New keywords for sharing data among program modules to allow for easy reference to VAX MACRO programs and VMS callable system routines

1.2.7 VAX COBOL

VAX COBOL is compatible with the ANSI-standard COBOL. It also supports an embedded data manipulation language (DML) interface to DIGITAL's CODASYL-compliant Database Management System. It allows access to common record definitions stored in the Common Data Dictionary. It is fully supported by the VMS operating system environment, including access to all utilities and the ability to invoke the callable system routines and to use object modules from other language programs.

VAX COBOL supports the following features:

- Full report-writing capabilities
- Form and report creation on terminals, with screen handling
- Complete sequential, relative, and indexed I/O
- All data types for ANSI COBOL, plus packed decimal, floating point, double floating point, and address data types
- Structured programming statements such as EVALUATE for CASE-like statement, scope-delimited statements to reduce use of GO TO, and inline PERFORM statements

1.2.8 VAX DIBOL

VAX DIBOL is designed specifically for interactive data processing. It includes a compiler, a debugger, and a set of utility programs that facilitate data handling, data storing, and interprogram communication. VAX DIBOL can invoke VMS Record Management Services (RMS), system services, utility routines, and run-time library routines. It can use object modules produced from any other VAX language program.

The VAX DIBOL compiler produces a source file listing, symbol table, label table, error report, error listing, and cross-reference listing. The VAX DIBOL Debugger Tool (DDT) allows you to examine or change program data at run time, to set breakpoints, and to examine the flow of execution. The other utilities include a VAX DIBOL Message Manager that stores and retrieves messages for VAX DIBOL programs and the VAX DIBOL Message Status that allows you to examine and delete any messages currently being held by the Message Manager.

1.2.9 VAX FORTRAN

VAX FORTRAN supports the ANSI-standard FORTRAN–77 and provides full support for many industry-standard FORTRAN features based on FORTRAN-66. It takes full advantage of the following VMS features:

- Supports all VMS RMS file formats
- Can access any object module generated by other languages
- Can create shareable images usable by any program written in a native language
- Has a high optimization compiler
- Can invoke all callable system routines
- Has record structure and Common Data Dictionary support
- Can use all programming utilities

1.2.10 VAX LISP

VAX LISP has a fully interactive interpreter and a compiler. VAX LISP can use many VMS resources including calling object modules written in any other VAX language, invoking VMS RMS and the other callable system routines, and using the VMS utilities. It includes its own debugger that enables examination of a running program, step execution, and traces.

1.2.11 VAX MACRO

VAX MACRO is an assembly language for programming the VAX computer under the VMS operating system. The instruction set includes approximately 130 instructions and 70 directives, which enable complex programming statements. It can use all VMS resources; it can invoke any callable system routine, use the VMS Symbolic Debugger and other utilities, and call any object module written in another VAX language.

General assembler directives can perform the following operations:

- Store data or reserve memory for data storage
- Control the alignment of parts of the program in memory
- Specify the methods of accessing the sections of memory in which the program will be stored
- Specify the entry point of the program or a part of the program
- Specify the way in which symbols are referenced
- Specify that a part of the program is to be assembled only under certain conditions
- Control the format and content of the listing file
- Display informational messages
- Control the assembler options that are used to interpret the source program
- Define new opcodes

VAX MACRO directives define macros and repeat blocks. With these directives, you can repeat identical or similar sequences of source statements and use string operators to manipulate and test the contents of source statements.

1.2.12 VAX PASCAL

VAX PASCAL takes full advantage of the VAX floating point hardware, character instructions sets, and virtual memory capability of the VMS operating system. VAX PASCAL can utilize all features of the VMS operating system, including the following:

- Support for the VMS Symbolic Debugger
- Compilation of separate modules
- Access to other object modules written in other languages
- Access to all callable system routines
- Access to Common Data Dictionary data declarations

Along with the standard ANSI Pascal features, VAX PASCAL incorporates the following features:

- Exponentiation and concatenation operator
- Hexadecimal, octal, and DOUBLE constants

- Uppercase and lowercase letters treated identically, except in character and string constants
- Dollar sign ($) and underscore (_) characters in identifiers
- DOUBLE, SINGLE, QUADRUPLE, VARYING character strings and UNSIGNED data types
- I/O, arithmetic, ordinal, boolean, transfer, dynamic allocation, character string manipulation, unsigned, and allocation size defined routines
- READ (or READLN) of user-defined ordinal type and string
- WRITE (or WRITELN) of user-defined scalar type or any data using binary, hexadecimal, or octal format
- Conformant array parameters for processing arrays with potentially different bounds
- Optional attribute specification on types, variables, routines, and compilation units in order to change many of the properties of a program

1.2.13 VAX PL/I

VAX PL/I incorporates the following features:

- A compile-time preprocessor that allows language extension and conditional compilation
- Several program control constructs (DO, IF-THEN-ELSE, BEGIN-END, LEVEL, SELECT-WHEN-OTHERWISE, and CALL)
- AUTOMATIC initializations, AREA (user allocation), OFFSET, scalar assignment to arrays, the REFER structure, the ENTRY statement, and the LIKE attribute
- Access to the Common Data Dictionary
- Symbolic Debugger support
- Access to callable system routines

1.2.14 VAX RPG II

VAX RPG II has full access to VMS resources, including the following:

- Use of VMS RMS and other callable system routines
- Full integration with the Symbolic Debugger
- Call object modules written in other languages
- Support of industry-standard RPG II specifications
- CALL extension on the calculation specification
- Automatic record matching and merging operations for multifile processing
- Multilevel control break handling
- Record identification codes

- Table and array processing
- Field editing features

1.2.15 VAX SCAN

VAX SCAN is a block-structured, high-level programming language that is designed to manipulate text strings and text files. The primary applications for VAX SCAN are filters, translators, extractors/analyzers, and preprocessors. Some of the features of VAX SCAN include the following:

- String operators for searching, comparing, extracting, and assigning character strings
- Matching of one or more complex patterns of text in the input data using SET,TOKEN, GROUP, and MACRO declarations
- Preprocessor that allows you to extend the language
- Symbolic Debugger support

1.3 Linker

After a program is compiled or assembled, it must be linked. The linker completes the following major steps:

- Resolves references.
- Combines multiple object files.
- Creates executable images (it assigns virtual address and produces an image map listing the addresses).

1.3.1 Using Options Files

The linker provides the mechanism for tying together object modules written in other VAX languages. The linker also can read a separate file (the options file) that lists long or complicated linker instructions so that you do not have to enter them on the command line.

1.3.2 Using Image Maps

You can also use the linker for debugging programs. Use the image map to locate an instruction that caused a run-time error, translate a number displayed by the debugger to its related symbol or address, and locate definitions of symbols.

1.3.3 Using LIBRARIAN with the Linker

Use LIBRARIAN to collect input—object modules, shareable images, and macros—for the linker. You can assign system-defined logical names to the libraries. Then, the linker automatically searches these libraries to resolve references. The library logical names are LNK$LIBRARY, LNK$LIBRARY_1 through LNK$LIBRARY_999. When you associate libraries with these logical names, do not skip any logical names in the sequence.

For complete information about using the VMS Linker, refer to the *VMS Linker Utility Manual.*

1.3.4 Linker Input and Output

Depending on the needs of your program, the linker can accept input from the following sources:

- Object file—Any object module created after compiling or assembling a source program.
- Shareable image file—A separate image that was already linked but which cannot be run as a separate file.
- Symbol table file—A separate symbol table produced by a previous linking operation. The symbol table contains global symbols and values of an image.
- Library file—A file containing one or more object modules and a symbol table of global symbols of each module, or one or more shareable images and a universal symbol table for each shareable image.
- Options file—Input file specifications and link options that cannot be defined at the DCL command level can be specified in this file.

Primarily, the linker produces an executable image of the program. In addition, the linker has the capability to produce the following:

- A shareable image—An image that can be used by other programs but cannot be executed independently.
- A system image—An image that does not execute under the control of the operating system; rather as a stand-alone operation on VAX hardware.
- An image map—A file containing additional program information including object module synopsis, module relocatable reference synopsis, image section synopsis, program section synopsis, symbols by name and value, image synopsis, and link run statistics.
- A symbol table file—A file containing symbols and their values to be used by other programs being linked.

1.3.5 Linker Command Summary

There are several linker command and positional qualifiers available. With these qualifiers, you can control linker operations in the following ways:

- To produce an abbreviated image map
- To generate a debug symbol table to give the debugger control when the image is run
- To place the entire executable image in P0 address space
- To produce and protect shareable images
- To create a system image
- To include traceback information in the image
- To seach system default, shareable image default, and user-default libraries to resolve references

1.3.6 Additional Features

The linker also incorporates the following features:

- An options file
- Image map
- Object language

Use an options file when you frequently use the same set of file specifications and file qualifiers, when you use a shareable image, and when you use more file and positional qualifiers than a DCL command can accommodate. There are also several link options available that can be specified only in the options files. These options include specifying a starting virtual address, controlling the order of object modules, specifying usage of a shareable image, renaming the image, specifying the number of pages for I/O, protecting access to a shareable image, allocating pages to the stack, assigning a value to a symbol, and converting a global symbol into a universal symbol.

The image map contains information on the contents of the image and on the link process. You can use the map to locate link-time errors, view the image layout in virtual memory, keep track of global symbols, and so forth.

Use VAX object language when you are writing compilers or assemblers that must generate object modules acceptable to the linker. The object language describes formats to which object files, library files, and symbol table files must conform. The object language defines an object module as an ordered set of variable-length records.

1.4 Debuggers

The two debugger utilities available with the VMS operating system are the VMS Symbolic Debugger (debugger) and the VMS Delta/XDelta Utility (DELTA/XDELTA). Use the debugger to debug user-mode code. Use Delta /XDelta to debug code in other modes. You can also use Delta to debug user-mode code, if you prefer.

For most programs, use the debugger. The debugger allows you to reference program locations using the symbols you defined in the program. You do not need to keep track of program addresses. Entering commands is easier with the debugger than with DELTA/XDELTA; you can use the keypad, the command line, or an input file to enter a lengthy series of commands. The debugger has a screen mode that allows you to view several lines of source code at one time, the commands you enter, and the output from the commands you enter. Error messages are more descriptive than the error message in DELTA/XDELTA, and help information is available.

1.4.1 Symbolic Debugger

You can use the debugger with the following programming languages: VAX Ada, VAX BASIC, VAX BLISS-32, VAX C, VAX COBOL, VAX DIBOL, VAX FORTRAN, VAX MACRO, VAX PASCAL, VAX PL/I, VAX RPG, and VAX SCAN. Detailed information and examples about how the debugger works and how to use the commands are provided in the *VMS Debugger Manual*.

The debugger provides the following features:

- Program locations can be referenced by the symbols you used in the source file. You do not need to use virtual addresses.
- All language data types are understood by the debugger. It displays program variables according to the declared data type.
- Data can be entered and displayed in several formats—ASCII, hexadecimal, octal, or decimal.
- You can set breakpoints at several points within a program: at a location, on certain types of events, on certain classes of instructions, or after each instruction.
- You can start or resume program execution in two ways: instruction-by-instruction or until either the next breakpoint or the end of the program.
- Program execution can be traced according to specified locations.
- Program execution can be suspended whenever a particular variable or other memory area has been modified.
- Program variables or locations can be examined and modified without having to leave the debugger, recompile, and relink the program.
- An expression (or address expression) can be evaluated during program execution.
- Debugger commands can be executed conditionally or executed repeatedly using FOR, IF, REPEAT, and WHILE control structures.
- Shareable images can be debugged.

- Initialization files can be set up to set default debugging modes, screen display definitions, keypad definitions, and symbol definitions specific to your debugging needs.
- Log files can be used to record each command during a debugging session. You can then use the log file as a command procedure for the next debugging session.
- You can define symbols to represent your own commands, address expressions, or values.

The debugger has over 100 commands available to control a debugging session. The commands are used for the following major debugging tasks:

- Control program execution on a line-by-line basis or at a breakpoint that you specify
- Display breakpoints, tracepoints, watchpoints, active routine calls, stack contents, variables, symbols, source code, and source directory search list
- Expand the debugger's memory pool
- Set screen mode and change which items are displayed on the screen and where items are displayed
- Define symbols
- Create key definitions
- Create and execute debugger command procedures
- Change values in variables
- Evaluate a language or address expression
- Change and show data types and the radix for data display

1.4.2 Delta/XDelta Utility

DELTA and XDELTA are debugging tools you can use to monitor the execution of user-mode programs, privileged-mode programs, and the VMS operating system. DELTA and XDELTA have the same commands and use the same expressions. However, they are different in two ways: you use them to debug different kinds of code, and you invoke and exit from them in different ways. For more information on using this utility for debugging, refer to the *VMS Delta/XDelta Utility Manual.*

You can use DELTA to debug user-mode programs or programs that execute at interrupt priority level (IPL) 0 in any processor mode. To run DELTA in a processor mode other than user mode, your process must have the privilege that allows DELTA to change to that mode—change-mode-to-executive (CMEXEC) or change-mode-to-kernel (CMKRNL) privilege. You cannot use DELTA to debug code that executes at an elevated IPL.

You can use XDELTA to debug programs that execute in any processor mode and at any IPL. To use XDELTA, you must be able to boot the system.

There are 19 DELTA/XDELTA commands that you can use to complete the following debugging tasks:

- Open, display, and change the value of a particular location
- Set, clear, and display breakpoints

- Set display modes in byte, word, longword, or ASCII
- Display instructions
- Execute the program in a single-step with the option to step over a subroutine
- Set base registers
- List the names and locations of all loaded modules of the executive

1.5 Programming Utilities

There are several programming utilities you can use in conjunction with program development. These utilities are described in this section. The most commonly used utilities are the Command Definition Utility, the Librarian Utility, and the Message Utility.

1.5.1 Command Definition Utility

The Command Definition Utility (CDU) creates DCL-level commands with syntax similar to DCL. You can modify your process command table, the system command table in SYS$LIBRARY, or create a new command table to be used with user-written applications.

1.5.1.1 Defining a New Command

With an editor, you create a command definition (CLD) file where you define the new command by specifying the following elements:

- Command verb
- Alternate syntax for the command
- Parameters, qualifiers, and keywords
- Type of data allowed as input to a parameter, qualifier, or keyword
- Allowable combinations of parameters, qualifiers, and keywords
- Allowable values for parameters, qualifiers, and keywords
- Object module information if you want the command definition to create an object module

1.5.1.2 Modifying the Process and System Command Tables

After you define a new command, you must add it to either your process command table or the system command table in SYS$LIBRARY using the DCL command SET COMMAND. Please note that you must have SYSPRV privilege to modify the DCL command table in SYS$LIBRARY. Modifications to the system command table affect all users.

1.5.1.3 Creating a New Command Table

When you set up a series of commands that can be used only within a user-written application, each command must be described in the command definition file. For each command, you must specify the name of a routine in a program that executes the command. Then, use the DCL command SET COMMAND/OBJECT to create an object module from the command definition file.

Link the command object module with your program.

1.5.1.4 Parsing the Command String

The program you write must be able to interpret any parameters, qualifers, or errors that are entered. In order to parse the commands, the program must use CDU's callable interface by including calls to command language interpreter (CLI) utility routines.

For more information about creating your own commands with CDU, refer to the *VMS Command Definition Utility Manual.*

Example 1–1 illustrates a command definition file that sets up a DCL-level command INCOME to invoke a program INCOME.EXE. You can select whether to enter the name of the database file, to specify an input file specification, or to generate a report.

Example 1–1 Defining a New Command

```
! Run INCOME.EXE when user types INCOME
DEFINE VERB INCOME
! Location of image
IMAGE DISK1:[INCOME]INCOME.EXE
! User can enter file name of database
PARAMTER P1
LABEL = STATS_FILE
VALUE (TYPE=$FILE, DEFAULT="STATS.SAV")
! User can type /ENTER
QUALIFIER ENTER
! User can type /FIX[=(value, ...)]
QUALIFIER FIX
VALUE (LIST)
! User can type /REPORT[=file-name]
QUALIFIER REPORT
VALUE (TYPE=$FILE, DEFAULT="INCOME.RPT")
! Can only do one thing
DISALLOW ANY2 (ENTER, FIX, REPORT)
```

1.5.2 Librarian Utility

Libraries are files you create to store frequently used modules of code or text. With the Librarian Utility (LIBRARIAN), you can create a library, maintain the modules in a library, or display information about a library and its modules. You use LIBRARIAN commands to manage modules within a library. You can use DCL commands to manage the entire library as one unit. For example, if you want to rename the library, use the DCL command RENAME.

1.5.2.1 Types of Libraries

There are several types of libraries. Table 1–1 lists the types of library that are available.

Table 1–1 Types of Libraries

	File Type		
Library	**Library**	**Module**	**Contents**
Help	HLB	HLP	Help text modules that provide users with information about a program
Macro	MLB	MAR	VAX MACRO definitions used as input to the assembler
Object	OLB	OBJ	Object modules of frequently called routines
Shareable image	OLB	EXE	Symbol tables of shareable images used as input to the linker
Text	TLB	TXT	Sequential record files used as input data to a program

1.5.2.2 Linking Your Program with Libraries

If you use LIBRARIAN to maintain large sets of object modules or macros, you can link a program with entire libraries using the DCL command LINK. For example, you could store 100 macros in the library INCOME.MLB. To link it with the program INCOME.OBJ, enter the following command:

```
$ LINK INCOME+INCOME.MLB/LIBRARY
```

1.5.2.3 Assigning Logical Names to Libraries

You can assign system-defined logical names to a library—LNK$LIBRARY_1 through LNK$LIBRARY_999. There are also 999 default logical names you can assign to a library—LNK$LIBRARY_1 through LNK$LIBRARY_999. When the linker attempts to resolve symbols, these libraries are searched automatically. If you assigned INCOME.MLB to LNK$LIBRARY_1, you would enter the following command to link the object module INCOME.OBJ.

```
$ LINK INCOME
```

1.5.2.4 Sharing Code Using Text Libraries

Sharing code can be easily accomplished by creating object or macro libraries that all users can access. You can also share data by creating text libraries of data files that all users can access.

For complete information on creating, managing, and using libraries, refer to the *VMS Librarian Utility Manual.*

1.5.2.5 Manipulating Libraries Using the LIBRARY Command

The DCL command LIBRARY invokes LIBRARIAN and accepts 28 command qualifiers. LIBRARIAN manages library modules in the following ways:

- Create a new library and specify the type
- Add, delete, or replace a module within the library
- Copy a module from the library
- List the modules in the library, with a history, with global symbols, or before or after a specified time
- Enables a log of each library action

You can create command procedures that manipulate libraries using the DCL command LIBRARY.

1.5.3 Message Utility

The Message Utility (MESSAGE) allows you to supplement the VMS system messages with your own messages. The message can signal any condition—error or success. Use an editor to create a message source file and compile it with MESSAGE. Then, link the message object module with the program object module. By using message pointers, you can use different text for the same message. This option is particularly useful for multilingual applications. To use pointers, you create a nonexecutable message file that contains the message text and a pointer file that contains the symbols and pointer to the nonexecutable file.

For complete information on creating your own messages, refer to the *VMS Message Utility Manual*. Refer to the *VMS System Messages and Recovery Procedures Reference Volume* for a list of the system messages and suggested recovery procedures.

The message source file consists of message definition statements and directives that define the message text, the message code values, and the message symbol. With these directives, you assign severity levels, specify message text, and define the facility to which the message relates.

Example 1–2 shows a message source file:

Example 1–2 Message Source File

```
.TITLE         SAMPLE Error and Warning Messages
.IDENT         'VERSION 4.00'
.FACILITY      SAMPLE,1/PREFIX=ABC_
.SEVERITY      ERROR

UNRECOG        < Unrecognized keyword !AS>/FAO_COUNT=1
AMBIG          < Ambiguous keyword>

.SEVERITY      WARNING
.BASE          10
SYNTAX         < Invalid syntax in keyword>

.END
```

1.5.4 Patch Utility

The Patch Utility (PATCH) allows you to make changes to an image file in the form of patches. You can then run the new version of the image without having to recompile (or reassemble) and relink the program. You can enter PATCH commands interactively or use them in a command procedure to execute interactively or in batch mode. You can use PATCH with any language supported by the VMS operating system as long as the image was generated by the linker.

For detailed information on how PATCH works and its commands, refer to the *VMS Patch Utility Manual*.

The input image can be a shareable image, a device driver image, or any other executable image. Consider the following restrictions when you use PATCH:

- You can specify only universal symbols when patching a shareable image.
- You can use the default patch area to patch position-independent shareable images.
- You must use a user-defined patch area to patch position-dependent images.

PATCH does not alter the input image. It creates a copy of the image, makes changes to the copy, and leaves the original image unaltered.

With the PATCH commands, you can modify the image as follows:

- Add instructions or data
- Delete instructions or data
- Replace instructions or data
- Allocate space for the patch area
- Create a command procedure of PATCH commands
- Assign an ECO-level to the changes
- View the contents of a particular location
- Display the modules in the image
- Apply the patch to the image

1.5.5 SUMSLP Utility

The SUMSLP Utility (SUMSLP) is a batch-oriented editor that is useful when you need to make several updates to a single file. To use it, you create a series of editing commands to add, delete, or update lines in the file. The editing commands are specific to SUMSLP and can be used only by SUMSLP. It can be useful if you are maintaining several copies of a single file, because it allows you to update the file by creating one update program and applying the update program to each copy of the file.

SUMSLP requires at least two input files as follows:

- The source file to be updated. Because you use line-oriented editing commands, you should generate a sequence-oriented listing.

- The update file. This file contains SUMSLP command lines and the updated lines used to alter the input file.

SUMSLP applies the edits specified in the SUMSLP update file to the input source file. The SUMSLP output file generated is the updated source file.

The *VMS SUMSLP Utility Manual* describes each of the SUMSLP commands and how SUMSLP processes files.

1.5.6 System Dump Analyzer

The System Dump Analyzer Utility (SDA) helps you determine the cause of system failures. You invoke this utility specifying a system crash dump file, which is a copy of memory at the time of a system crash. SDA reads the dump file; then, it formats and displays the contents of the file. In addition to information contained in the dump file, SDA reads the system's symbol table file. You can specify that SDA read the symbols that define many of the system's data structures, including those in the I/O database.

You can also use SDA to analyze a running system. To do this, you need change-mode-to-kernel (CMKRNL) privilege. This option is useful for examining the stack and memory of a process stalled in a scheduler state.

If you are examining a dump file, SDA displays the immediate cause of the crash. You can then use SDA to diagnose how the error occurred. For example, you can use SDA commands to locate the line of code that signaled the bugcheck and to find the line of code (usually on the stack) that caused the error. Then, you can examine device drivers, linker maps, and system maps to locate the module where the line of code came from. Once the module has been identified, you can examine the module code to pinpoint the problem.

You can locate the error using SDA commands that allow you to view the following pieces of information:

- The location and contents of the four process stacks and the systemwide interrupt stack
- The active processes and the values of the parameters used in swapping and scheduling these processes
- The software and hardware context of any process
- The value of a symbol and the contents of the location the symbol points to
- A formatted list of a block of memory
- The list of system page table entries
- The look-aside pools, the nonpaged dynamic storage pool, and the paged dynamic storage pool
- All locks in the system
- The names of the VMS RMS data structures
- All data structures associated with a device
- The VAX cluster or the system communications services cluster

- The active connections between systems communication services processes
- The dump file header
- The response identifications

The SDA commands also allow you to switch processes, direct output to a log file or terminal, scan memory locations, assign a value to a symbol, read global symbols to add them to the SDA symbol table, and repeat the execution of the last command.

1.5.7 National Character Set Utility

The National Character Set Utility (NCS) allows you to define and use collating sequences and conversion functions. With collating sequences, you can alter the standard sorting sequence for a particular use (usually for a national character set). Using conversion functions, you can define case conversions or character representations that you subsequently use in the collating sequence.

The collating sequences and conversions are stored in an NCS library that you manage using NCS. The command qualifiers allow you to create the library; insert, replace, and delete modules; list module information; and view specified modules.

Eight NCS callable routines allow you to access the collating sequences and conversions stored in an NCS library from your program.

Refer to the *VMS National Character Set Utility Manual* for more information about using NCS and its callable routines.

1.6 Callable System Routines

There are four sets of callable system routines as follows:

- Run-time library (RTL) routines
- System services
- Utility routines
- VMS RMS

You can use the system routines in your program to complete a number of programming tasks, including the following:

- I/O operations
- Security procedures
- File manipulation
- Memory management
- Screen management
- Mathematics operations
- Event synchronization
- Utility usage

The sections that follow suggest sets of routines to use for each of these general programming tasks. For the specific routines to use, refer to Sections 1.6.9, 1.6.10, 1.6.11, and 1.6.12.

1.6.1 I/O Operations

For I/O operations, you can use VMS RMS, RTL routines, or system services. Use VMS RMS for device-independent I/O, when you want more control over file access. Use RTL routines to get more functionality than language I/O statements. Use system services for device-dependent I/O when you want more control over the device. System services allow you to access devices not supported by VMS RMS, to perform I/O operations not supported by a particular language, and to increase I/O performance.

1.6.2 Security Procedures

For security procedures, use system services to maintain rights database, to use access control lists and process rights lists, to check access protection, and to provide security erase patterns. To assign protection to a particular file, use VMS RMS.

1.6.3 File Management

For complex file manipulation, you would generally use the VMS Record Management Services. VMS RMS can create complex file organizations; reorganize files; extend disk space for files; and get, locate, insert, update, and delete records. There are VMS RMS and RTL routines for simple file manipulation such as opening, reading, deleting, renaming, and closing files.

1.6.4 Memory Management

For memory management tasks, both RTL routines and system services can acquire and free virtual memory. RTL memory management routines call system services. RTL routines maintain a processwide pool of free pages that are automatically reused. If you call system services directly, the program must keep track of free pages. Direct calls to system services should be used when the size requirements exceed 1000 pages for one request. RTL routines working with such large requests may result in fragmenting the virtual address space. System services give you more control because you can specify a specific virtual address and unlock pages in memory.

1.6.5 Screen Management

For screen management, use RTL routines. The screen management routines allow you to build terminal-independent screen management functions. They do not rely on particular hardware devices; input is read from a virtual keyboard and output is sent to a virtual display. With SMG$ routines, complex screens can be built with several regions defined. The program can then work within a region without regard to its position on the screen.

1.6.6 Math Operations

For math routines, RTL routines can complete simple arithmetic as well as the following functions:

- Exponentiation
- Complex exponentiation
- Complex function evaluation
- Floating-point trigonometric function evaluation
- Absolute value
- Numeric data conversions

1.6.7 Event Synchronization

For event synchronization, use RTL routines or system services. Use RTL routines to synchronize events with event flags. Use system services to synchronize events with event flags, with a resource lock, and with an asynchronous system trap (AST).

1.6.8 Calling Utility Routines

To access VMS utilities in a program, you use the utility routines. These routines allow your program to call several VMS utilities and to use the utility either interactively or noninteractively.

1.6.9 Run-Time Library (RTL) Routines

The RTL routines are grouped into six categories according to programming task as follows:

- DECtalk (DTK$) routines

  Use DTK$ routines for manipulating DIGITAL's DECtalk devices.

- General purpose Run-Time Library (LIB$) routines

  Use LIB$ routines for general programming tasks such as memory management, file management, VAX MACRO instruction calls, I/O operations, condition handling, cross-referencing, DCL command interpreter operations, and synchronization. These routines are listed in Table 1–2.

- Mathematics (MTH$) routines

  Use MTH$ routines for completing common arithmetic, algebraic, and trigonometric functions.

- General purpose (OTS$) routines

  Use OTS$ routines for extensive data-type conversion and some computation.

- Parallel processing (PPL$) routines

  Use PPL$ routines for parallel processing of program units.

- Screen management (SMG$) routines

  Use SMG$ routines to perform terminal-independent screen management. These routines are summarized in Table 1–3.

- String manipuation (STR$) routines

  Use STR$ routines for string manipulation of fixed-length, variable-length, and dynamic strings.

For a complete description of using the RTL for these tasks, as well as a routine-by-routine description, refer to the *VMS Run-Time Library Routines Volume*.

Table 1–2 RTL General Programming Tasks (LIB$) Routines

General Systems Tasks		
LIB$ADAWI	LIB$CURRENCY	LIB$DATE_TIME
LIB$DAY	LIB$DAY_OF_WEEK	LIB$DEC_OVER
LIB$DIGIT_SEP	LIB$DO_COMMAND	LIB$ENABLE_CTRL
LIB$FID_TO_NAME	LIB$FIXUP_FLT	LIB$FLT_UNDER
LIB$GET_COMMAND	LIB$GET_FOREIGN	LIB$GET_SYI
LIB$INT_OVER	LIB$LOOKUP_KEY	LIB$LP_LINES
LIB$PAUSE	LIB$RUN_PROGRAM	LIB$SUBX
I/O Operations		
LIB$ASN_WTH_MBX	LIB$GETDVI	LIB$GET_INPUT
LIB$PUT_OUTPUT	LIB$SYS_FAO	LIB$SYS_FAOL
LIB$SYS_GETMSG		
Queue		
LIB$GETQUI	LIB$INSQHI	LIB$INSQTI
LIB$REMQHI	LIB$REMQTI	
Synchronization and Event Flags		
LIB$AST_IN_PROG	LIB$FREE_EF	LIB$GET_EF
LIB$RESERVE_EF	LIB$WAIT	

Table 1–2 (Cont.) RTL General Programming Tasks (LIB$) Routines

Time Functions		
LIB$ADD_TIMES	LIB$CONVERT_DATE_STRING	LIB$CVT_FROM_INTERNAL_TIME
LIB$CVTF_FROM_INTERNAL_TIME	LIB$CVT_TO_INTERNAL_TIME	LIB$CVTF_TO_INTERNAL_TIME
LIB$CVT_VECTIM	LIB$FORMAT_DATE_TIME	LIB$FREE_DATE_TIME_CONTEXT
LIB$FREE_TIMER	LIB$GET_DATE_FORMAT	LIB$GET_MAXIMUM_DATE_LENGTH
LIB$GET_TIMER	LIB$GET_USERS_LANGUAGE	LIB$INIT_DATE_TIME_CONTEXT
LIB$MULT_DELTA_TIME	LIB$MULTF_DELTA_TIME	LIB$SHOW_TIMER
LIB$STAT_TIMER	LIB$SUB_TIMES	LIB$SYS_ASCTIM
Process Control		
LIB$ATTACH	LIB$GETJPI	LIB$SPAWN
File Management		
LIB$CREATE_DIR	LIB$DELETE_FILE	LIB$FILE_SCAN
LIB$FILE_SCAN_END	LIB$FIND_FILE	LIB$FIND_FILE_END
LIB$RENAME_FILE	LIB$TRIM_FILESPEC	
Logical Names and Symbols		
LIB$DELETE_LOGICAL	LIB$DELETE_SYMBOL	LIB$FIND_IMAGE_SYMBOL
LIB$FREE_LUN	LIB$GET_LUN	LIB$GET_SYMBOL
LIB$RADIX_POINT	LIB$SET_LOGICAL	LIB$SET_SYMBOL
LIB$SYS_TRNLOG		
VAX MACRO Instructions		
LIB$BBCCI	LIB$BBSSI	LIB$CALLG
LIB$CRC	LIB$CRC_TABLE	LIB$EDIV
LIB$EMODx	LIB$EMUL	LIB$EXTV
LIB$EXTZV	LIB$FFC	LIB$FFS
LIB$INSV	LIB$MOVC3	LIB$MOVC5
LIB$MOVTC	LIB$MOVTUC	LIB$POLYz
LIB$SPANC		
Data Type Conversion		
LIB$CHAR	LIB$CVT_DX_DX	LIB$CVT_xTB

Table 1–2 (Cont.) RTL General Programming Tasks (LIB$) Routines

String Management		
LIB$GET_COMMON	LIB$INDEX	LIB$LEN
LIB$LOCC	LIB$MATCHC	LIB$PUT_COMMON
LIB$SCANC	LIB$SCOPY_DXDX	LIB$SCOPY_R_DX
LIB$SFREE1_DD	LIB$SFREEN_DD	LIB$SGET1_DD
LIB$TRA_ASC_EBC	LIB$TRA_EBC_ASC	LIB$TPARSE
Memory Management		
LIB$CREATE_USER_VM_ZONE	LIB$CREATE_VM_ZONE	LIB$DELETE_VM_ZONE
LIB$FIND_VM_ZONE	LIB$FREE_VM	LIB$GET_VMJ
LIB$GET_VM_PAGE	LIB$RESET_VM_ZONE	LIB$SHOW_VM
LIB$SHOW_VM_ZONE	LIB$STAT_VM	
Cross-referencing		
LIB$CRF_INS_KEY	LIB$CRF_INS_REF	LIB$CRF_OUTPUT
Condition Handling		
LIB$DECODE_FAULT	LIB$DISABLE_CTRL	LIB$ESTABLISH
LIB$MATCH_COND	LIB$REVERT	LIB$SIGNAL
LIB$SIG_TO_STOP	LIB$SIM_TRAP	LIB$STOP
Binary Trees		
LIB$INSERT_TREE	LIB$LOOKUP_TREE	LIB$TRAVERSE_TREE

Table 1–3 RTL Screen Management (SMG$) Routines

General Routines	
SMG$CREATE_SUBPROCESS	SMG$DELETE_SUBPROCESS
SMG$DEL_TERM_TABLE	SMG$EXECUTE_COMMAND
SMG$GET_NUMERIC_DATA	SMG$GET_TERM_DATA
SMG$INIT_TERM_TABLE	SMG$INIT_TERM_TABLE_BY_TYPE

Table 1–3 (Cont.) RTL Screen Management (SMG$) Routines

Input Routines	
SMG$ADD_KEY_DEF	SMG$CANCEL_INPUT
SMG$CHANGE_VIEWPORT	SMG$CREATE_KEY_TABLE
SMG$CREATE_MENU	SMG$CREATE_VIRTUAL_KEYBOARD
SMG$CREATE_VIEWPORT	SMG$DEFINE_KEY
SMG$DELETE_KEY_DEF	SMG$DELETE_MENU
SMG$DELETE_VIEWPORT	SMG$DELETE_VIRTUAL_KEYBOARD
SMG$GET_KEY_DEF	SMG$GET_KEYBOARD_ATTRIBUTES
SMG$GET_VIEWPORT_CHAR	SMG$LIST_KEY_DEFS
SMG$LIST_PASTING_ORDER	SMG$LOAD_KEY_DEFS
SMG$READ_COMPOSED_LINE	SMG$READ_KEYSTROKE
SMG$READ_STRING	SMG$READ_VERIFY
SMG$REPLACE_INPUT_LINE	SMG$RETURN_INPUT_LINE
SMG$SET_DEFAULT_STATE	SMG$SET_KEYBOARD_MODE
SMG$SCROLL_VIEWPORT	SMG$SELECT_FROM_MENU
SMG$SET_TERM_CHARACTERISTICS	

Output Routines	
SMG$BEGIN_DISPLAY_UPDATE	SMG$BEGIN_PASTEBOARD_UPDATE
SMG$CHANGE_PBD_CHARACTERISTICS	SMG$CHANGE_RENDITION
SMG$CHANGE_VIRTUAL_DISPLAY	SMG$CHECK_FOR_OCCLUSION
SMG$CONTROL_MODE	SMG$COPY_VIRTUAL_DISPLAY
SMG$CREATE_PASTEBOARD	SMG$CREATE_VIRTUAL_DISPLAY
SMG$CURSOR_COLUMN	SMG$CURSOR_ROW
SMG$DELETE_CHARS	SMG$DELETE_LINE
SMG$DELETE_PASTEBOARD	SMG$DELETE_VIRTUAL_DISPLAY
SMG$DISABLE_BROADCAST_TRAPPING	SMG$DISABLE_UNSOLICITED_INPUT
SMG$DRAW_CHAR	SMG$DRAW_LINE
SMG$DRAW_RECTANGLE	SMG$ENABLE_UNSOLICITED_INPUT
SMG$END_DISPLAY_UPDATE	SMG$END_PASTEBOARD_UPDATE
SMG$ERASE_CHARS	SMG$ERASE_COLUMN
SMG$ERASE_DISPLAY	SMG$ERASE_LINE
SMG$ERASE_PASTEBOARD	SMG$FIND_CURSOR_DISPLAY
SMG$FLUSH_BUFFER	SMG$GET_BROADCAST_MESSAGE
SMG$GET_CHAR_AT_PHYSICAL_CURSOR	SMG$GET_DISPLAY_ATTR
SMG$GET_PASTEBOARD_ATTRIBUTES	SMG$GET_PASTING_INFO
SMG$GET_PHYSICAL_CURSOR	SMG$HOME_CURSOR

Table 1–3 (Cont.) RTL Screen Management (SMG$) Routines

Output Routines	
SMG$INSERT_CHARS	SMG$INSERT_LINE
SMG$INVALIDATE_DISPLAY	SMG$LABEL_BORDER
SMG$MOVE_TEXT	SMG$MOVE_VIRTUAL_DISPLAY
SMG$PASTE_VIRTUAL_DISPLAY	SMG$POP_VIRTUAL_DISPLAY
SMG$PRINT_PASTEBOARD	SMG$PUT_CHARS
SMG$PUT_CHARS_HIGHWIDE	SMG$PUT_CHARS_WIDE
SMG$PUT_CHARS_MULTI	SMG$PUT_HELP_TEXT
SMG$PUT_LINE	SMG$PUT_LINE_HIGHWIDE
SMG$PUT_LINE_MULTI	SMG$PUT_LINE_WIDE
SMG$PUT_PASTEBOARD	SMG$PUT_STATUS_LINE
SMG$PUT_VIRTUAL_DISPLAY_ENCODED	SMG$READ_FROM_DISPLAY
SMG$REPAINT_LINE	SMG$REPAINT_SCREEN
SMG$REPASTE_VIRTUAL_DISPLAY	SMG$RESTORE_PHYSICAL_SCREEN
SMG$RETURN_CURSOR_POS	SMG$RING_BELL
SMG$SAVE_PHYSICAL_SCREEN	SMG$SCROLL_DISPLAY_AREA
SMG$SET_BROADCAST_TRAPPING	SMG$SET_CURSOR_ABS
SMG$SET_CURSOR_MODE	SMG$SET_CURSOR_REL
SMG$SET_DISPLAY_SCROLL_REGION	SMG$SET_OUT_OF_BAND_ASTS
SMG$SET_PHYSICAL_CURSOR	SMG$SNAPSHOT
SMG$UNPASTE_VIRTUAL_DISPLAY	

1.6.10 System Services

System services are routines used by the operating system to complete several tasks. Programs can use the following system services to complete similar actions for an individual program:

- Security services work with rights databases, access control lists, and security erase patterns. They can also be called to check access to files and magnetic tape.
- Event flag services clear, set, and read event flags.
- Event synchronization services enable, call, and disable asynchronous event traps (AST) to synchronize events.
- Logical name services associate and disassociate logical names with physical devices and maintain logical name tables.
- Input/output services control I/O devices directly to maximize I/O efficiency.
- Process control services create, delete, and control processes.
- Timer services can schedule program events and obtain and format binary time values.

- Condition handling services specify particular routines to assume control after a hardware or software exception condition occurs.
- Memory management services manipulate virtual address space for a program, control paging and swapping, and create and access files in memory containing shareable code or data.
- Change mode services change the mode of a process
- Lock management services synchronize access to shared resources.

The system services are summarized according to their functions in Table 1–4.

Table 1–4 Summary of System Services

Service	Function
System Security Services	
$ADD_HOLDER	Adds holder record to rights database
$ADD_IDENT	Adds identifier to rights database
$ASCTOID	Translates identifier name to binary value
$CHANGE_ACL	Creates or modifies an ACL
$CHECK_ACCESS	Invokes system access protection check on behalf of another user
$CHKPRO	Invokes system access protection check
$CMEXEC	Change to executive mode
$CMKRNL	Change to kernal mode
$CREATE_RDB	Initializes a rights database
$ERAPAT	Generates a security erase pattern
$FIND_HELD	Returns identifiers held by a holder in rights database
$FIND_HOLDER	Returns holders of an identifier in rights database
$FINISH_RDB	Deallocates record stream and clears context value when searching the rights database
$FORMAT_ACL	Formats ACE into a text string
$GETUAI	Get user authorization information
$GRANTID	Adds identifier to process or system rights list
$IDTOASC	Translates identifier value to its identifier name
$MOD_HOLDER	Modifies holder record in rights database
$MOD_IDENT	Modifies identifier record in rights database
$MTACCESS	Controls magnetic tape access
$PARSE_ACL	Converts text ACE into binary format
$REM_HOLDER	Deletes holder record from identifier's list of holders in rights database
$REM_IDENT	Deletes identifier and all holders of that identifier from rights database

Table 1–4 (Cont.) Summary of System Services

Service	Function
System Security Services	
$REVOKID	Removes identifier from process or system rights list
$SETDFPROT	set default file protection
$SETSSF	set system services filter
$SETUAI	set user authorization information
Event Flag Services	
$ASCEFC	Associate common event flag
$DACEFC	Disassociate common event flag
$DLCEFC	Delete common event flag
$SETEF	Set common event flag
$CLREF	Clear event flag
$READEF	Read event flag
$WAITFR	Wait for single event flag
$WFLOR	Wait for logical OR of event flags
$WFLAND	Wait for logical AND of event flags
$ENQ and $ENQW	Enqueue lock request
Synchronization services	
$SETAST	Set AST enable
$DCLAST	Declare AST
$SETPRA	Set power recovery AST
$SYNCH	Synchronize
Logical Name Services	
$CRELNM	Create logical name
$CRELNT	Create logical name table
$DELLNM	Delete logical name
$TRNLNM	Translate logical name table

Table 1–4 (Cont.) Summary of System Services

Service	Function
I/O Services	
$ASSIGN	Assign I/O channel
$DASSIGN	Deassign I/O channel
$QIO	Queue I/O request
$QIOW	Queue I/O request and wait
$FAO	Formatted ASCII output
$FAOL	Formatted ASCII output with list parameter
$ALLOC	Allocate device
$DALLOC	Deallocate device
$MOUNT	Mount volume
$DISMOU	Dismount volume
$GETDVI	Get device and channel information
$GETDVIW	Get device and channel information and wait
$GETQUI	Get queue information
$GETQIOW	Get queue information and wait
$CANCEL	Cancel I/O on channel
$CREMBX	Create mailbox and assign channel
$DELMBX	Delete mailbox
$BRKTH	Breakthrough
$BRKTHW	Breakthrough and wait
$SNDJBC	Send message to job controller
$SNDJBCW	Send message to job controller and wait
$SNDOPR	Send message to operator
$SNDERR	Send message to error logger
$GETMSG	Get message
$PUTMSG	Put message

Table 1–4 (Cont.) Summary of System Services

Service	Function
Control Processes Services	
$CREPRC	Create process
$DELPRC	Delete process
$SUSPND	Suspend process
$RESUME	Resume process
$HIBER	Hibernate
$WAKE	Wake
$SCHDWK	Schedule wakeup
$CANWAK	Cancel wakeup
$EXIT	Exit
$FORCEX	Force exit
$DCLEXH	Declare exit handler
$CANEXH	Cancel exit handler
$SETPRN	Set process name
$SETPRI	Set priority
$SETPRV	Set privileges
$SETRWM	Set resource wait mode
$GETJPI	Get job/process
Timer Services	
$GETTIM	Get time
$NUMTIM	Convert binary time to numeric time
$ASCTIM	Convert binary time to ASCII string
$BINTIM	Convert ASCII string to binary time
$SETIMR	Set timer
$CANTIM	Cancel timer request
$SETTIME	Set system time
Condition Handler Services	
$SETEXV	Set exception vector
$SETSFM	Set system service failure exception mode
$UNWIND	Unwind from condition handler frame
$DCLCMH	Declare change mode or compatibility mode handler

Table 1–4 (Cont.) Summary of System Services

Service	Function
Memory Management Services	
$EXPREG	Expand program/control region
$CNTREG	Contract program/control region
$CRETVA	Create virtual address space
$DELTVA	Delete virtual address space
$CRMPSC	Create and map section
$MGBLSC	Map global section
$DGBLSC	Delete global section
$UPDSEC	Update section file on disk
$LKWSET	Lock pages in working set
$ULWSET	Unlock pages from working set
$ADJWSL	Adjust working set limit
$PURGWS	Purge working set
$LCKPAG	Lock page in memory
$UNLPAG	Unlock page in memory
$SETPRT	Set protection on pages
$SETSWM	Set process swap mode
$SETSTK	Set stack limits
Lock Request Services	
$ENQ	Enqueue lock request
$DEQ	Dequeue lock request
$GETLKI	Get lock information
File Management Services	
$FILESCAN	Scan string for file specification
$RMSRUNDWN	RMS rundown
$SETDDIR	Set default directory

1.6.11 Utility Routines

Certain VMS utilities provide a callable interface that can be accessed from programs. Utility routines provide access from within a program to several VMS utilities. The utility routines allow the program to invoke the utility, execute utility-specific functions, and exit the utility. Some VMS utilities can be invoked at the DCL-command level or through a callable interface. Other utilities have only a callable interface. Table 1–5 summarizes the utility routine groups.

For complete information on the utility routines, and a routine-by-routine listing, refer to the *VMS Utility Routines Manual.*

Table 1–5 Utility Routine Summary

Routine Prefix	Utility/Facility	Description
ACL$	Access Control List (ACL) Editor	Creates and maintains access control lists. ACLs controls access to files, devices, global sections, logical name tables, or mailboxes.
CLI$	Command Definition Utility (CDU)	Processes command strings using information from a command table; use in conjunction with new commands created by CDU.
CONV$	Convert and Convert /Reclaim (CONV) Utility	Convert utility copies records from one or more files to an output file while changing format and file organization. Convert/reclaim utility reclaims empty buckets so that new records can be written.
DCX$	Data Compression /Expansion (DCX) Facility	Analyzes and compresses data records; expands data records that have been compressed.
EDT$	EDT (EDT) Editor	Invokes EDT and either edits a file from the program, or allows interactive editing.
FDL$	File Definition Language Utility (FDL)	Specifies VMS RMS options for a file, creates a file, opens a file, closes a file, connects a file, allocates VMS RMS control blocks, fills in control blocks, and deallocates control blocks.
LBR$	Librarian Utility (LBR)	Maintains any type of library.
MAIL$	Mail Utility (MAIL)	Sends mail messages to users on the system or any connected system from an application program.
PSM$	Print Symbiont Modification (PSM) Facility	Modifies the VMS print symbiont (or, if necessary, can be used to create user-written symbiont).
SMB$	Symbiont/Job-Controller Interface (SMB) Facility	Provides the symbiont-job controller interface for user-written symbionts.
SOR$	Sort/Merge (SOR) Utility	Integrates a sort or merge opertion into a program application.
TPU$	VAX Text Processing Utility (VAXTPU)	Invokes and uses VAXTPU functions within a program written in any VAX programming language.

1.6.12 VMS Record Management Services

VMS RMS assists user programs in processing and managing files and their contents. VMS RMS allows you to create a new file, access an existing file, extend disk space for a file, close a file, obtain file characteristics as well as to get, locate, insert, update, and delete records.

VMS RMS provides the following items:

- Disk file organizations—Sequential, relative, and indexed
- Record formats—Fixed length and variable length for each file organization

- Record access modes—Sequential, by key value, by relative record number, by record file address

For complete information about using VMS RMS, refer to the *VMS Record Management Services Manual.*

1.6.12.1 Device Support

VMS RMS supports unit-record devices such as terminals and printers, but it is designed primarily to provide a comprehensive software interface to mass-storage devices such as disk and magnetic tape drives.

1.6.12.2 VMS RMS File Control Blocks

Control blocks are used to provide input to services and to accept output from services.

The following control blocks support services that manipulate files:

- File access block (FAB)

  The FAB control block includes file specification information, file characteristics (file organization, record type, allocation information, and so forth), and run-time access options (file processing information and address(es) of other control blocks with additional information.)

- Optional name block (NAM)

  The NAM control block includes supplemental information to the FAB.

- Optional extended attribute block (XAB)

  The XAB control block includes file characteristics that supersede or supplement the FAB control block.

1.6.12.3 VMS RMS Record Control Blocks

To support services that manipulate with records, there are two record control blocks, as follows:

- Record access block (RAB)

  The RAB control block includes the address of the related FAB control block, the address of input and output record buffers, general I/O buffer type and size, how the records will be accessed, and other record information.

- Extended attribute block (XAB)

  The XAB control block includes record characteristics that can supersede or supplement information in the RAB control block.

1.6.12.4 VMS RMS Macros

VMS RMS uses macros provided in the system macro library to perform the following tasks:

- Initialize control blocks at assembly time (allocates space within the program image for the control block, defines the symbolic names for a control block, initializes certain control block fields with internally used values, initializes specified control block fields with user-specified values, and initializes certain fields with system-supplied default values).
- Define control block symbolic names at assembly time (does not allocate or initialize the control block).
- Set specified fields with user-specified values at run time.
- Invoke services at run time.

Table 1–6 lists each control block and its associated macros.

Table 1–6 User Control Blocks

Control Block	Macro Name	Function
FAB		Describes a file and contains file-related information.
	$FAB	Allocates storage for a FAB and initializes certain FAB fields; also defines symbolic offsets for a FAB.
	$FABDEF	Defines symbolic offsets for a FAB.
	$FAB_STORE	Moves specified values into a previously allocated and initialized FAB.
NAM		Contains file specification information beyond that in the file access block.
	$NAM	Allocates storage for a NAM and initializes certain NAM fields; also defines symbolic offsets for a NAM.
	$NAMDEF	Defines symbolic offsets for a NAM.
	$NAM_STORE	Moves specified values into a previously specified and allocated NAM.
RAB		Describes a record stream and contains record-related information.
	$RAB	Allocates storage for a RAB and initializes certain RAB fields; also defines symbolic offsets for a RAB.
	$RABDEF	Defines symbolic offsets for a RAB.
	$RAB_STORE	Moves specified values into a previously specified and allocated RAB.

Table 1–6 (Cont.) User Control Blocks

Control Block	Macro Name	Function
XABxxx[1]		Contains file attribute information beyond that in the file access block. For XABTRM, contains information beyond that in the record access block.
	$XABxxx	Allocates and initializes an XAB.
	$XABxxxDEF	Defines symbolic offsets for an XABxxx.
	$XABxxx_STORE	Moves specified values into a previously specified and allocated XABxxx.

[1]The xxx is a 3-character mnemonic.

1.6.13 VMS Record Management Services Utilities

The following are the three VMS RMS utilities:

- Analyze/RMS_File Utility (ANALYZE/RMS_FILE)
- Convert and Convert/Reclaim Utilities (CONVERT and CONVERT /RECLAIM)
- Create/FDL Utility (CREATE/FDL)
- Edit/FDL Utility (EDIT/FDL)

You can use these independently of VMS RMS, or in conjunction with VMS RMS, to build data files and to maintain files.

1.6.13.1 ANALYZE/RMS_FILE

With ANALYZE/RMS_FILE, you can analyze the internal structure of an VMS RMS file in the following manner:

- Examine the structure of a file, and interactively check the structure to assess if it is properly designed for the application
- Generate a statistical report on the file's structure and use
- Generate an FDL file from a data file
- Generate a summary report on the file's structure and use

The interactive feature of this utility includes several commands to traverse the structure of an VMS RMS file and examine specific data buckets and bytes of a record. This utility can also check the file and generate a report listing any errors found in the file. Refer to the *VMS Analyze/RMS_File Utility Manual*.

ANALYZE/RMS_FILE commands help you move around the VMS RMS file easily. You can move the structure pointer to the beginning and end of the file structure, up and down levels, to the first and last nodes, and to a specific bucket (or record) of an indexed or relative file.

1.6.13.2 CONVERT and CONVERT/RECLAIM

CONVERT copies one or more records from a file to another file, while changing the record format and file organization. CONVERT/RECLAIM reclaims empty bucket space in the file to allow new records to be written to it.

CONVERT/RECLAIM works only with Prolog 3 indexed files. You should use CONVERT/RECLAIM when new records no longer need a primary key associated with the deleted record.

In conjunction with changing record format and file organization, you can use CONVERT to complete the following functions:

- Reformat indexed files where many records have been inserted and deleted. New record file addresses are established for the records.
- Create a new output file with the same or different file characteristics.
- Add new records to the end of an existing sequential file.
- Merge new records into an existing indexed file.
- Convert carriage control to one of four formats (CARRIAGE_RETURN, FORTRAN, PRINT, and NONE).

CONVERT/RECLAIM does not change file format or organization when it reclaims empty bucket space. It deletes the old pointers to a bucket and puts it on a list of free buckets. When new records that need a new bucket are added, VMS RMS goes to the free bucket list and sets up pointers to a bucket from the list. CONVERT/RECLAIM preserves record's file addresses.

For a complete description of using CONVERT and CONVERT/RECLAIM, refer to the *VMS Convert and Convert/Reclaim Utility Manual*.

Command qualifiers allow you to modify CONVERT in the following ways:

- Append records to an existing file
- Create a new file with or without using an FDL file
- Access or insert records in an indexed file
- Pad short records or truncate long records
- Sort a file according to the primary key
- Check all read and write operations

1.6.13.3 CREATE/FDL and EDIT/FDL

The File Definition Language (FDL) helps you define specifications for data files. FDL is used within the context of the File Definition Language Facility and consists of the utilities CREATE/FDL and EDIT/FDL. An FDL file consists of a collection of file attributes grouped into related sections. EDIT /FDL invokes the FDL editor to create a new FDL file. The types of attributes you specify are the following:

- File processing operations specified using the following keywords: BLOCK_IO enabling VMS RMS read and write operations, DELETE, GET, PUT, RECORD_IO enabling mixed record I/O and block I/O TRUNCATE, UPDATE)
- Allocation of area and key analysis sections (for indexed files only)

- Creation or manipulation of VMS RMS-specific areas in an indexed file
- Application-dependent run-time attributes
- Date and time for certain file characteristics
- File processing and file-related characteristics
- Key attributes
- Secondary attributes that define records specified using the following keywords: BLOCK_SPAN, CARRIAGE_CONTROL, CONTROL_FIELD, FORMAT, and SIZE
- Sharing of the data file
- System identification information

CREATE/FDL uses the specifications in an existing FDL file to create a new empty data file. The *VMS File Definition Language Facility Manual* describes how to use the FDL utility and lists each of the commands.

With FDL commands, you can add, modify, or delete lines to a file; enable assistance with the design and optimization of a data file; specify the number of keys in an indexed file; specify the output file; divide an indexed file into a specified number of areas; and choose between smaller buffer and flatter files.

1.7 System Programming

System programming includes the following types of tasks:

- Writing your own system services
- Writing your own print symbiont or modifying existing ones
- Writing your own device driver
- Writing code that requires privileged access mode
- Writing code that operates at an elevated interrupt priority level (IPL)

To write code that operates at a system level, special practices must be followed to ensure that your work does not corrupt the operating system or other system-level code. For example, device driver routines do not run sequentially from beginning to end; therefore, device driver code must follow standard VMS conventions to ensure proper synchronization.

Information for system programmers is contained in the following documents:

- *VMS System Services Volume*
- *VMS I/O User's Reference Volume*
- *VMS Device Support Manual*
- *VAX MACRO and Instruction Set Reference Manual*
- *VMS Delta/XDelta Utility Manual*

2 Using Processes

A process is the environment where an image executes. You can create and manage processes to complete the following programming tasks:

- Modularize application programs to have a single process executing a single function
- Dedicate a process to execute DCL commands
- Perform parallel processing where one process executes one part of a program while another process executes another part
- Implement application program management where one process manages and coordinates the activities of several other processes
- Schedule program execution
- Isolate code for one or more of the following reasons:
  - Debug logic errors
  - Execute privileged code
  - Execute sensitive code

2.1 Creating Processes

A created process can be either a *spawned* subprocess or a *detached* process.

2.1.1 Types of Processes

A spawned subprocess is dependent on the process that created it (its parent) and is deleted when the parent process exits. A detached process is independent of the process that created it. If you want a created process to continue after the parent exits, use a detached process. You can also use detached processes to write to another process's terminal (use the system sevice SYS$BREAKTHRU).

2.1.2 Modes of Execution

A process executes in one of the following modes:

- Interactive—Receives input from a record-oriented device such as a terminal or mailbox
- Batch—Created by the job controller and not interactive
- Network—Created by the network ACP
- Other—A process not running in any of the other modes (for example, a spawned subprocess where input is received from a command procedure)

Table 2–1 summarizes the characteristics of detached processes versus subprocesses.

Table 2–1 Detached Processes and Spawned Subprocesses

Characteristic	Spawned	Detached
Privileges	From creating process	Specified by creating process
Quotas and limits	Shared with creating process	Specified by creating process, but not shared with creating process
User Authorization File	Used for information not given by creating process	Used for most information not given by creating process
User Identification Code	UIC of creating process	Specified by creating process
Restrictions	Exists as long as creating process exists	None
How created	SYS$CREPRC or LIB$SPAWN from another process	SYS$CREPRC from another process
When deleted	At image exit, or when creating process exits	At image exit
Command Language Interpreter present	Usually not	Usually not

2.1.3 Creating Spawned Subprocesses

You can create a spawned subprocess using LIB$SPAWN, SYS$CREPRC, or PPL$CREATE_PROCESS.

2.1.3.1 Creating a Spawned Subprocess Using LIB$SPAWN

Because LIB$SPAWN is designed specifically for spawned processes, by default, it provides more context values for the subprocess than SYS$CREPRC. For example, LIB$SPAWN creates a subprocess with the same priority as the parent process (generally 4).

LIB$SPAWN allows you to create a subprocess and set some context options for the new subprocess. The format for LIB$SPAWN is:

```
LIB$SPAWN ([command_string],[input_file]
,[output_file],[flags],[process-name],[process_id],[completion_status]
,[completion_efn],[completion_astadr],[completion_astarg],[prompt],[cli])
```

For complete information on using each argument, refer to the LIB$SPAWN routine in *VMS Run-Time Library Routines Volume*.

Specifying a Command String

Specify a single DCL command to execute once the subprocess is initiated using the **command_string** argument. You can also use this argument to execute a command procedure to execute several DCL commands (@command_procedure_name).

Redefining SYS$ERROR, SYS$INPUT, and SYS$OUTPUT

Specify alternate input, output, and error devices for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR using the **error**, **input**, and **output** arguments. Using alternate values for SYS$INPUT, SYS$OUTPUT and SYS$ERROR can be particularly useful when you are synchronizing processes that are executing concurrently.

Passing Parent Process Context Information to the Subprocess

Specify which characteristics of the parent process are to be passed onto the subprocess using the **flags** argument. With this argument, you can reduce the time required to create a subprocess by passing only a part of the parent's context. You can also specify whether the parent process should continue to execute (execute concurrently) or wait until the subprocess has completed execution (execute in line).

After the Subprocess Completes Execution

Specify the action to be taken when the subprocess completes execution (send a completion status, set a local event flag, or invoke an AST procedure) using the **completion_**status, **completion_efn**, and **completion_astadr** arguments. For more information on event flags and ASTs, refer to Chapter 4.

Specifying an Alternate Prompt String

Specify a different prompt string for the subprocess using the **prompt** argument.

Specifying an Alternate Command Language Interpreter

Specify a different command language interpreter for the subprocess using the **cli** argument.

2.1.3.2 Creating a Spawned Subprocess Using SYS$CREPRC

With SYS$CREPRC, you must usually specify the priority because the default priority is zero. Though SYS$CREPRC does not set many context values for the subprocess by default, it does allow you to set many more context values than LIB$SPAWN. For example, you cannot specify separate privileges for a subprocess with LIB$SPAWN directly, but you can with SYS$CREPRC.

By default, SYS$CREPRC creates a subprocess rather than a detached process. The format for SYS$CREPRC is as follows:

SYS$CREPRC ([pidadr],[image],[input],[output],[error],[prvadr],[quota] ,[prcnam], [baspri], [uic] ,[mbxunt],[stsflg])

For a complete description of how to use each argument, refer to the *VMS System Services Reference Manual.*

The default values passed onto the subprocess might not be complete enough for your use. Use the SYS$CREPRC to modify these default values as described below.

Redefining SYS$ERROR, SYS$INPUT, and SYS$OUTPUT

Specify alternate input, output, and error devices for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR using the **error**, **input**, and **output** arguments. Using alternate values for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR can be particularly useful when you are synchronizing processes that are executing concurrently.

Setting Privileges

Set different privileges for the subprocess using the **prvadr** argument. This is particularly useful when you want to dedicate a subprocess to execute privileged or sensitive code.

Setting Process Quotas

Set different process quotas of system resources for the subprocess using the **quota** argument. This option can be useful when managing a subprocess to limit use of system resources (such as AST usage, I/O, CPU time, lock requests, and working set size and expansion).

Setting the Subprocess Priority Level

Set the subprocess priority using the **baspri** argument. If you do not set it, the default value is 0.

Specifying Additional Processing Options

Enable and disable parent and subprocess wait mode, control process swapping, control process accounting; control process dump information; control authorization checks; and control working set adjustments using the **stsflg** argument.

2.1.3.3 Creating a Spawned Subprocess Using PPL$CREATE_PROCESS

PPL$CREATE_PROCESS works similarly to LIB$SPAWN in that it creates subprocesses with the same context as the parent process. In addition, you can create more than one subprocess at a time, and you can specify the name of an image to be executed in the subprocess. However, you should limit use of PPL$CREATE_PROCESS to creating subprocesses specifically for parallel processing.

Before using PPL$CREATE_PROCESS, you must set up special PPL$ data structures with the PPL$INITIALIZE routine; otherwise, unpredictable results may occur. Also, after you create a process with PPLS$CREATE_PROCESS, you should delete it with PPL$DELETE_PROCESS.

PPL$CREATE_PROCESS creates one or more subprocesses on the same node (or system) as the parent process. The format for this routine is:

PPL$CREATE_PROCESS ([number-of-processes],[image-name], [process-vector].[flags])

For complete information on using each argument, refer to the *VMS RTL Parallel Processing (PPL$) Manual.*

Specifying the Number of Subprocesses

Specify the number of subprocesses to be created using the **number-of-processes** argument. If no value is specified, one subprocess is created.

Specifying the Name of the Image

Specify the name of the image to be run in the new subprocess using the **image-name** argument. If no name is provided, the image being run in the parent process is invoked.

Specifying Processing Options

Specify one of two options for the new subprocesses: (1) the new process is not part of a parallel application or (2) the new subprocess executes the program without the VMS Symbolic Debugger.

Table 2-2 lists the context values provided by LIB$SPAWN and SYS$CREPRC for a subprocess when using the default values in the routine calls.

Table 2-2 Comparison of LIB$SPAWN, SYS$CREPRC, and PPL$CREATE_PROCESS

Context	LIB$SPAWN	SYS$CREPRC	PPL$CREATE_PROCESS
DCL	Yes	No[1]	Yes
Default device and directory	Parent's	Parent's	Parent's
Symbols	Parent's	No	Parent's
Logical Names	Parent's[2]	No[2]	Parent's[2]
Privileges	Parent's	Parent's	Parent's
Priority	Parent's	0	Parent's

[1]The created subprocess can include DCL by executing the system image SYS$SYSTEM:LOGINOUT.EXE (example in Section 2.1.4.).

[2]Plus group and job logical name tables.

Review the features of each routine in the following manuals to determine which routine is the best for your needs:

- LIB$SPAWN—*VMS RTL Library (LIB$) Manual*
- SYS$CREPRC—*VMS System Services Reference Manual*
- PPL$CREATE_PROCESS— *VMS RTL Parallel Processing (PPL$) Manual*

2.1.3.4 Debugging Within a Subprocess

Another option you might consider is to allow a program to be debugged within a subprocess. To allow debug operations, equate the subprocess logical names DBG$INPUT and DBG$OUTPUT to the terminal. When the subprocess executes the program, which has been compiled and linked with the debugger, the debugger reads input from DBG$INPUT and writes output to DBG$OUTPUT.

If you are executing the subprocess concurrently, you should restrict debugging to the program in the subprocess. The debugger prompt *DBG>* should enable you to differentiate between input required by the parent process and input required by the subprocess. However, each time the debugger displays information, you must press the RETURN key to display the *DBG>* prompt. (By pressing the RETURN key, you actually write to the parent process, which has regained control of the terminal following the subprocess's writing to the terminal. Writing to the parent process allows the subprocess to regain control of the terminal.)

Examples of Creating Subprocesses

The following example creates a subprocess that executes the commands in COMMANDS.COM, which must be a command procedure on the current default device in the current default directory. The created subprocess inherits symbols, logical names (including SYS$INPUT and SYS$OUTPUT), keypad definitions, and other context information from the parent. The subprocess executes while the parent process hibernates.

```
! Declare status and library routine
INTEGER STATUS, LIB$SPAWN

STATUS = LIB$SPAWN ('@COMMANDS')
```

The following program segment creates a subprocess that does not inherit the parent's symbols, logical names, or keypad definitions. The subprocess reads and executes the commands in the command procedure COMMANDS.COM. (The CLI$ symbols are defined in the $CLIDEF module of the system object or shareable image library; see Section 5.3.3.)

```
! Mask for LIB$SPAWN
INTEGER MASK
EXTERNAL CLI$M_NOCLISYM,
2        CLI$M_NOLOGNAM,
2        CLI$M_NOKEYPAD
! Declare status and library routine
INTEGER STATUS, LIB$SPAWN

! Set mask and call LIB$SPAWN
MASK = %LOC(CLI$M_NOCLISYM) .OR.
2      %LOC(CLI$M_NOLOGNAM) .OR.
2      %LOC(CLI$M_NOKEYPAD)

STATUS = LIB$SPAWN ('@COMMANDS.COM',
2                   ,,
2                   MASK)
```

The following program segment creates a subprocess to execute the image $DISK1:[USER.MATH]CALC.EXE. CALC reads data from DATA84.IN and writes the results to DATA84.RPT. The subprocess executes concurrently. (CLI$M_NOWAIT is defined in the $CLIDEF module of the system object or shareable image library; see Section 5.3.3.)

```
! Mask for LIB$SPAWN
EXTERNAL CLI$M_NOWAIT
! Declare status and library routine
INTEGER STATUS, LIB$SPAWN

STATUS = LIB$SPAWN ('RUN $DISK1:[USER.MATH]CALC', ! Image
2                   'DATA84.IN',                 ! Input
2                   'DATA84.RPT',                ! Output
2                   %LOC(CLI$M_NOWAIT))          ! Concurrent
```

2.1.4 Creating Detached Processes

In general, create a detached process only when a program must continue executing after the parent process exits. DETACH privilege is required for most detached process operations. Use the system service SYS$CREPRC.

Examples of Creating a Detached Process

The following program segment creates a process that executes the image SYS$USER:[ACCOUNT]INCTAXES.EXE. INCTAXES reads input from the file TAXES.DAT and writes output to the file TAXES.RPT. (TAXES.DAT and TAXES.RPT are in the default directory on the default disk.) The last argument specifies that the created process is a detached process (the UIC defaults to that of the parent process). (The symbol PRC$M_DETACH is defined in the $PRCDEF module of the system macro library.)

```
EXTERNAL  PRC$M_DETACH

! Declare status and system routines
INTEGER STATUS,SYS$CREPRC
    .
    .
    .
STATUS = SYS$CREPRC (,
2                    'SYS$USER:[ACCOUNT]INCTAXES',  ! Image
2                    'TAXES.DAT',                   ! SYS$INPUT
2                    'TAXES.RPT',                   ! SYS$OUTPUT
2                    ,,,,
2                    %VAL(4),                       ! Priority
2                    ,,
2                    %VAL(%LOC(PRC$M_DETACH))       ! Detached
```

The following program segment creates a detached process to execute the DCL commands in the file SYS$USER:[TEST]COMMANDS.COM. The system image SYS$SYSTEM:LOGINOUT.EXE is executed to include DCL in the created process. The DCL commands to be executed are specified in a command procedure that is passed to SYS$CREPRC as the input file. Output is written to the file SYS$USER:[TEST]OUTPUT.DAT.

```
    .
    .
    .
STATUS = SYS$CREPRC (,
2                    'SYS$SYSTEM:LOGINOUT',         ! Image
2                    'SYS$USER:[TEST]COMMANDS.COM',! SYS$INPUT
2                    'SYS$USER:[TEST]OUTPUT.DAT',   ! SYS$OUTPUT
2                    ,,,,
2                    %VAL(4),                       ! Priority
2                    ,,
2                    %VAL(%LOC(PRC$M_DETACH))       ! Detached
```

2.2 Managing Processes

Managing a process includes the following programming tasks:

- Obtaining process information
- Setting process privileges
- Setting process priority
- Controlling process swapping
- Hibernating or suspending a process
- Setting process name
- Deleting a process
- Synchronizing process execution

You can use system routines and DCL commands to accomplish these tasks. Table 2–3 summarizes which routines and commands to use. You can use the DCL commands in a command procedure file that is executed as soon as the subprocess (or detached process) is created.

For process synchronization techniques, refer to Chapter 4.

Table 2–3 Routines and Commands for Managing Processes

Routine	DCL Command	Task
LIB$GETJPI SYS$GETJPI SYS$GETJPIW	SHOW PROCESS	Return process information
SYS$SETPRV	SET PROCESS	Set process privileges
SYS$SETPRI	SET PROCESS	Set process priority
SYS$SETSWM	SET PROCESS	Control swapping of process
SYS$HIBER SYS$SUSPND SYS$RESUME	SET PROCESS	Hibernate and suspend process
SYS$SETPRN	SET PROCESS	Set process name
SYS$EXIT SYS$FORCEX SYS$DELPRC	EXIT and STOP	Delete process

By default, the routines and commands reference the current process. To reference another process, you must specify either the process identification number (PID) or the process name when you call the routine or with a command qualifier when you enter commands. You must have GROUP privilege to reference a process with the same group number and a different member number in its UIC and WORLD privilege to reference a process with a different group number in its UIC.

The information presented in this section covers using the routines. If you want to use the DCL commands in a command procedure, refer to the *VMS DCL Dictionary*.

2.2.1 Obtaining Process Information

You can use any of the three GETJPI routines to obtain information about processes. The differences among these routines are as follows:

- SYS$GETJPI operates asynchronously.
- SYS$GETJPIW and LIB$GETJPI operate synchronously.
- SYS$GETJPI and SYS$GETJPIW can obtain one or more pieces of information about processes in a single call.
- LIB$GETJPI can obtain only one piece of information about processes in a single call.
- SYS$GETJPI and SYS$GETJPIW can specify an AST to execute at the completion of the routine.
- SYS$GETJPI and SYS$GETJPIW can use an I/O Status Block (IOSB) to test for completion of the routine.
- LIB$GETJPI can return some items as strings or as numbers.

The *VMS Run-Time Library Routines Volume* and *VMS System Services Reference Manual* contain complete descriptions of these routines including a complete listing of all the items of information that you can request. LIB$GETJPI, SYS$GETJPI, and SYS$GETJPIW share the same item list with the following exception: LIB$K_ items can be accessed only by LIB$GETJPI.

In the following example, the string argument rather than the numeric argument is specified, causing LIB$GETJPI to return the UIC of the current process as a string:

```
! Define request codes
INCLUDE '($JPIDEF)'

! Variables for LIB$GETJPI
CHARACTER*9 UIC
INTEGER LEN

STATUS = LIB$GETJPI (JPI$_UIC,
2                    ,,,
2                    UIC,
2                    LEN)
```

If you want to get the same information about each process on the system, specify the process identification argument as -1 when you invoke LIB$GETJPI or SYS$GETJPI(W). Call the GETJPI routine (whichever you choose) repetitively until it returns a status of SS$_NOMOREPROC, indicating that all processes on the system have been examined.

Example 2–1 creates a file, PROCNAME.RPT, that lists, using LIB$GETJPI, the process name of each process on the system. If the process running this program does not have the privilege necessary to access a particular process, the program writes the words NO PRIVILEGE in place of the process name. If a process is suspended, LIB$GETJPI cannot access it and the program writes the word SUSPENDED in place of the process name. Note that, in either of these cases, the program changes the error value in STATUS to a success value so that the loop calling LIB$GETJPI continues to execute.

Example 2–1 Obtaining the Process Name

```
    .
    .
    .
! Status variable and error codes
INTEGER STATUS,
2       STATUS_OK,
2       LIB$GET_LUN,
2       LIB$GETJPI
INCLUDE '($SSDEF)'
PARAMETER (STATUS_OK = 1)

! Logical unit number and file name
INTEGER*4 LUN
CHARACTER*(*) FILE_NAME
PARAMETER (FILE_NAME = 'PROCNAME.RPT')
! Define item codes for LIB$GETJPI
INCLUDE '($JPIDEF)'

! Process name
CHARACTER*15 NAME
INTEGER LEN
! Process identification
INTEGER PID /-1/
    .
    .
    .
! Get logical unit number and open the file
STATUS = LIB$GET_LUN (LUN)
OPEN (UNIT = LUN,
2     FILE = 'PROCNAME.RPT',
2     STATUS = 'NEW')
! Get information and write it to file
DO WHILE (STATUS)
  STATUS = LIB$GETJPI(JPI$_PRCNAM,
2                     PID,
2                     ,,
2                     NAME,
2                     LEN)
  ! Extra space in WRITE commands is for
  ! FORTRAN carriage control
  IF (STATUS) THEN
    WRITE (UNIT = LUN,
2          FMT = '(2A)') ' ', NAME(1:LEN)
    STATUS = STATUS_OK
  ELSE IF (STATUS .EQ. SS$_NOPRIV) THEN
    WRITE (UNIT = LUN,
2          FMT = '(2A)') ' ', 'NO PRIVILEGE'
    STATUS = STATUS_OK
  ELSE IF (STATUS .EQ. SS$_SUSPENDED) THEN
    WRITE (UNIT = LUN,
2          FMT = '(2A)') ' ', 'SUSPENDED'
    STATUS = STATUS_OK
  END IF

END DO
```

Example 2–1 Cont'd. on next page

Example 2–1 (Cont.) Obtaining the Process Name

```
! Close file
IF (STATUS .EQ. SS$_NOMOREPROC)
2 CLOSE (UNIT = LUN)
   .
   .
   .
```

To specify a list of items for SYS$GETJPI or SYSGETJPIW (even if that list contains only one item), use a record structure. Example 2–2 uses SYS$GETJPIW to request the process name and user name associated with the process whose process identification number is in SUBPROCESS_PID.

Example 2–2 Obtaining Different Types of Process Information

```
   .
   .
   .
! PID of subprocess
INTEGER SUBPROCESS_PID

! Include the request codes
INCLUDE '($JPIDEF)'
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Declare GETJPI itmlst
RECORD /ITMLST/ JPI_LIST(3)
! Declare buffers for information
CHARACTER*15    PROCESS_NAME
CHARACTER*12    USER_NAME
INTEGER*4       PNAME_LEN,
2               UNAME_LEN
! Declare I/O status structure
STRUCTURE /IOSB/
 INTEGER*2 STATUS,
2          COUNT
 INTEGER*4 %FILL
END STRUCTURE
! Declare I/O status variable
RECORD /IOSB/ JPISTAT
```

Example 2–2 Cont'd. on next page

Example 2–2 (Cont.) Obtaining Different Types of Process Information

```
! Declare status and routine
INTEGER*4       STATUS,
2               SYS$GETJPIW
                      .
                      . ! Define SUBPROCESS_PID
                      .
! Set up itmlst
JPI_LIST(1).BUFLEN    = 15
JPI_LIST(1).CODE      = JPI$_PRCNAM
JPI_LIST(1).BUFADR    = %LOC(PROCESS_NAME)
JPI_LIST(1).RETLENADR = %LOC(PNAME_LEN)
JPI_LIST(2).BUFLEN    = 12
JPI_LIST(2).CODE      = JPI$_USERNAME
JPI_LIST(2).BUFADR    = %LOC(USER_NAME)
JPI_LIST(2).RETLENADR = %LOC(UNAME_LEN)
JPI_LIST(3).END_LIST  = 0
! Request information and wait for it
STATUS = SYS$GETJPIW (,
2                     SUBPROCESS_PID,
2                     ,
2                     JPI_LIST,
2                     JPISTAT,
2                     ,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Check final return status
IF (.NOT. JPISTAT.STATUS) THEN
  CALL LIB$SIGNAL (%VAL(JPISTAT.STATUS))
END IF
    .
    .
    .
```

2.2.2 Setting Privileges

Use the SYS$SETPRV system service to set process privileges. Setting process privileges allows you to limit executing privileged code to a specific process, to limit functions within a process, and to limit access from other processes. You can either enable or disable a set of privileges and assign privileges on a temporary or permanent basis. To use this service, the creating process must have the appropriate privileges.

2.2.3 Scheduling Processes

To alter the system's process scheduling, you can change the base priority of a process and lock a process into physical memory so that it is not swapped out. Processes with higher priority levels, or those that have been locked, are executed first.

If you create a subprocess with the LIB$SPAWN routine, you can set the priority of the subprocess by executing the DCL command SET PROCESS /PRIORITY as the first command in a command procedure. You can also use the SYS$SETPRI system service to change the priority of any process, regardless of how you created it. You must have ALTPRI privilege to increase a process's base priority above the base priority of the creating process.

If you create a subprocess with the LIB$SPAWN routine, you can inhibit swapping by executing the DCL command SET PROCESS/NOSWAP as the first command in a command procedure. Inhibit swapping for any process with the SYS$SETSWM system service. A process must have PSWAPM privilege to inhibit swapping.

Altering process scheduling must be done with care. Review the following considerations before you attempt to alter the standard process scheduling with either SYS$SETPRI or SYS$SETSWM:

- Priority—Increasing a process's base priority gives that process more processor time at the expense of processes executing at lower priorities. This is not recommended unless you have a program that must respond immediately to events (for example, a real-time program). If you must increase your base priority, return it to normal as soon as possible. If the entire image must execute at an increased priority, reset the base priority before exiting; image termination does not reset the base priority.
- Swapping—Inhibiting swapping keeps your process in physical memory. This is not recommended unless the effective execution of your image depends on it (for example, if the image executing in the process is collecting statistics on processor performance).

2.2.4 Changing Process Names

Use the system service SYS$SETPRN to change a process name. Changing process names might be useful when a lengthy image is being executed. You can change names at significant points in the program; then monitor program execution through the change in process names. You can obtain a process name with a GETJPI routine from within a controlling process; with a CONTROL+T keystroke if the image is currently executing in your process; or a DCL command SHOW SYSTEM if the program is executing in a detached process.

The following program segment calculates the tax status for a number of households, sorts the households according to tax status, and writes the results to a report file. Since this is a time-consuming process, the program changes the process name at major points so that progress can be monitored.

```
   .
   .
   .
! Calculate approximate tax rates
STATUS = SYS$SETPRN ('INCTAXES')
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = TAX_RATES (TOTAL_HOUSES,
2                   PERSONS_HOUSE,
2                   ADULTS_HOUSE,
2                   INCOME_HOUSE,
2                   TAX_PER_HOUSE)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Sort
STATUS = SYS$SETPRN ('INCSORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = TAX_SORT (TOTAL_HOUSES,
2                  TAX_PER_HOUSE)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
```

```
! Write report
STATUS = SYS$SETPRN ('INCREPORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
    .
    .
    .
```

2.2.5 Controlling Process Execution

You can control process execution in the following ways:

- Suspending a process
- Hibernating a process
- Stopping a process
- Resuming a process

Suspending or hibernating a process puts it into a dormant state; the process is not deleted, but the image within it is not being executed. A process in hibernation can control itself; a process in suspension requires another process to control it. Refer to Table 2–4 for a comparison of processes in hibernation versus suspension.

Table 2–4 Comparison of Hibernation and Suspension

Hibernation	Suspension
Can only hibernate self	Can suspend self or others
ASTs can be delivered	ASTs can only be queued
Can wake itself	Cannot resume itself
Can schedule a wakeup	Cannot schedule resume
Hibernate/wake is fast	Suspend/resume is slower
Requires little system overhead	Requires system dynamic memory

The following table summarizes the routines available to suspended and hibernating processes.

Routine	Function
Hibernating Processes	
SYS$HIBER	Places a process in hibernation
SYS$WAKE	Resumes execution of a process in hibernation
SYS$SCHDWK	Resumes execution of a process in hibernation at a specified time
LIB$WAIT	Places a process in hibernation for a specified number of seconds
SYS$CANWAK	Cancels a scheduled wake-up issued by SYS$SCHDWK

Routine	Function
Suspended Processes	
SYS$SUSPEND	Places a process in a suspended state
SYS$RESUME	Resumes execution of a process in a suspended state

2.2.6 Deleting Processes

You can use one of the following system services to delete a subprocess or a detached process. Some services terminate execution of the image in the process; others terminate the process itself.

- SYS$EXIT—Initiates normal exit in the current image. Control returns to the command interpreter. If there is no command interpreter, the process is terminated. This routine cannot be used to terminate an image in a detached process.
- SYS$FORCEX—Initiates a normal exit on the image in the specified process. GROUP and WORLD privilege may be required, depending on the process specified. An AST is sent to the specified process. The AST calls on the SYS$EXIT routine to complete the image exit. Because an AST is used, you cannot use this routine on a suspended process. You can use this routine on a subprocess or detached process.
- SYS$DELPRC—Deletes the specified process. GROUP or WORLD privilege may be required depending on the process specified. A termination message is sent to the calling process's mailbox. You can use this routine on a subprocess, a detached process, or the current process.

3 Communication

The VMS operating system allows your process to communicate with itself, with other processes, with the system, and with other systems. This chapter describes how these levels of communication can be used to perform the following functions:

- To synchronize events
- To share data
- To obtain information about events important to the program you are executing

3.1 Communicating Within a Process

Communicating within a process, from one program component to another, can be performed using the following methods:

- Local event flags
- Logical names (in supervisor mode)
- Global symbols
- Common blocks

Local event flags, logical names, symbols, and common blocks are equally acceptable for passing information among "chained" images since the image reading the information executes immediately after the image that deposited it. Only common blocks allow you to reliably pass data from one image to another with a separate image executing in between. However, event flags, logical names, and symbols can be extremely useful for communicating within a single image. Further, event flags are useful for synchronizing events within a single image.

Since permanent mailboxes and permanent global sections are not deleted when the creating image exits, they also could be used to pass information from the current image to a later executing image. However, use of the common block is recommended since it uses fewer system resources than the permanent structures and does not require privilege. (You need PRMMBX privilege to create a permanent mailbox and PRMGBL privilege to create a permanent global section.)

3.1.1 Local Event Flags

Event flags are status posting bits maintained by the VMS operating system for general programming use. Programs can set, clear, and read event flags. By setting and clearing event flags at specific points, one program component can signal when an event has occurred. Other program components can then check the event flag to determine when the event has been completed. For more information on using local and common event flags for synchronizing events, refer to Chapter 4.

3.1.2 Logical Names

Logical names can store up to 255 bytes of data. When you need to pass information from one program to another within a process, you can assign data to a logical name when you create the logical name; then, other programs can access the contents of the logical name.

3.1.2.1 Using Logical Name Tables

If both processes are part of the same job, you can place the logical name in the process logical name table (LNM$PROCESS) or in the job logical name table (LNM$JOB). If a subprocess is prevented from inheriting the process logical name table, you must communicate using the job logical name table. If the processes are in the same group, place the logical name in the group logical name table LNM$GROUP (requires GRPNAM or SYSPRV privilege). If the processes are not in the same group, place the logical name in the system logical name table LNM$SYSTEM (requires SYSNAM or SYSPRV privilege). Symbols can also be used, but only between a parent and a spawned subprocess that has inherited the parent's symbols.

3.1.2.2 Access Modes

A logical name can be created under any three access modes—user, supervisor, or executive. If you create a logical name in user mode, it is deleted after the image exits. If you create a logical name in supervisor or executive mode, it is retained after the image exits. Therefore, to share data within the process from one image to the next, use supervisor-mode logical names.

3.1.2.3 Creating and Accessing Logical Names

Perform the following steps to create and access a logical name:

1. Create the logical name and store data in it. Use LIB$SET_LOGICAL to create a supervisor logical name. No special privileges are required. You can also use the system service SYS$CRELNM, but you need SYSNAM privilege to create a supervisor logical name. SYS$CRELNM also allows you to create a logical name for the system or group table and create a logical name in any other mode, assuming you have appropriate privileges.
2. Access the logical name. Use the routine LIB$SYS_TRNLOG or SYS$TRNLNM. LIB$SYS_TRNLOG calls SYS$TRNLNM to search for the logical name and return information about it.
3. Once you have finished using the logical name, delete it. Using the routine LIB$DELETE_LOGICAL or SYS$DELLNM. LIB$DELETE_LOGICAL deletes the supervisor logical name without requiring any special privileges. SYS$DELLNM requires special privileges to delete logical names for privileged modes. However, you can also use this

routine to delete logical name tables or a logical name within a system or group table.

Example 3–1 creates a spawned subprocess to perform an iterative calculation. The logical name REP_NUMBER specifies the number of times that REPEAT, the program executing in the subprocess, should perform the calculation. Since both the parent process and the subprocess are part of the same job, REP_NUMBER is placed in the job logical name table LNM$JOB. (Note that logical names are case sensitive; specifically, LNM$JOB is a system-defined logical name that refers to the job logical name table, whereas lnm$job is not.) To satisfy the references to LNM$_STRING, include the file $LNMDEF.

Example 3–1 Creating a Spawned Subprocess

```
PROGRAM CALC

! Status variable and system routines
INTEGER*4 STATUS,
2         SYS$CRELNM,
2         LIB$GET_EF,
2         LIB$SPAWN
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Declare itmlst
RECORD /ITMLST/ LNMLIST(2)
! Number to pass to REPEAT.FOR
CHARACTER*3 REPETITIONS_STR
INTEGER REPETITIONS
! Symbols for LIB$SPAWN and SYS$CRELNM
EXTERNAL CLI$M_NOLOGNAM,
2        CLI$M_NOCLISYM,
2        CLI$M_NOKEYPAD,
2        CLI$M_NOWAIT,
2        LNM$_STRING
                 .
                 . ! Set REPETITIONS_STR
                 .
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.) Creating a Spawned Subprocess

```
! Set up and create logical name REP_NUMBER in job table
LNMLIST(1).BUFLEN     = 3
LNMLIST(1).CODE       = LNM$_STRING
LNMLIST(1).BUFADR     = %LOC(REPETITIONS_STR)
LNMLIST(1).RETLENADR  = 0
LNMLIST(2).END_LIST   = 0
STATUS = SYS$CRELNM (,
2                     'LNM$JOB',      ! Logical name table
2                     'REP_NUMBER',,  ! Logical name
2                     LNMLIST)        ! List specifying
                                      ! equivalence string
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Execute REPEAT.FOR in a subprocess
MASK = %LOC (CLI$M_NOLOGNAM) .OR.
2      %LOC (CLI$M_NOCLISYM) .OR.
2      %LOC (CLI$M_NOKEYPAD) .OR.
2      %LOC (CLI$M_NOWAIT)
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$SPAWN ('RUN REPEAT',,,MASK,,,,FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
   .
   .
   .
```

REPEAT.FOR

```
PROGRAM REPEAT
! Repeats a calculation REP_NUMBER of times,
! where REP_NUMBER is a logical name
! Status variables and system routines
INTEGER STATUS,
2       SYS$TRNLNM,
2       SYS$DELLNM
! Number of times to repeat
INTEGER*4   REITERATE,
2           REPEAT_STR_LEN
CHARACTER*3 REPEAT_STR
! Item list for SYS$TRNLNM
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.) Creating a Spawned Subprocess

```
! Declare itmlst
RECORD /ITMLST/ LNMLIST (2)
! Define item code
EXTERNAL LNM$_STRING
! Set up and translate the logical name REP_NUMBER
LNMLIST(1).BUFLEN    = 3
LNMLIST(1).CODE      = LNM$_STRING
LNMLIST(1).BUFADR    = %LOC(REPEAT_STR)
LNMLIST(1).RETLENADR = %LOC(REPEAT_STR_LEN)
LNMLIST(2).END_LIST  = 0
STATUS = SYS$TRNLNM (,
2                     'LNM$JOB',      ! Logical name table
2                     'REP_NUMBER',,  ! Logical name
2                     LNMLIST)        ! List requesting
                                      ! equivalence string
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Convert equivalence string to integer
! BN causes spaces to be ignored
READ (UNIT = REPEAT_STR (1:REPEAT_STR_LEN),
2     FMT = '(BN,I3)') REITERATE
! Calculations
DO I = 1, REITERATE
   .
   .
   .
END DO
! Delete logical name
STATUS = SYS$DELLNM ('LNM$JOB',       ! Logical name table
2                     'REP_NUMBER',) ! Logical name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

3.1.3 Command Language Interpreter Symbols

The symbols you create and access for process communication are command language interpreter (CLI) symbols. These symbols are stored in symbol tables maintained for use within the context of DCL, the default command language interpreter. They can store up to 255 bytes of information. The use of these symbols is limited to processes using DCL. If the process is not using DCL, an error status is returned by the symbol routines.

For information on using symbols within your program to share data, refer to Chapter 5.

3.1.3.1 When to Use Global Symbols

When you need to pass information from one program to another within a process, you can assign data to a global symbol when you create the symbol. Then, other programs can access the contents of the global symbol. You should use global symbols so that the value within the symbol can be accessed by programs.

3.1.3.2 When to Use Local Symbols

If you use a local symbol, only DCL-level commands can access the symbol.

3.1.4 Creating and Using Global Symbols

To use DCL global symbols, follow this procedure:

1. Create the symbol and assign data to it using the routine LIB$SET_SYMBOL. Make sure you specify that the symbol will be placed in the global symbol table in the **tbl-ind** argument. If you do not specify the global symbol table, the symbol will be a local symbol.
2. Access the symbol with the LIB$GET_SYMBOL routine. This routine uses DCL to return the value of the symbol as a string.
3. Once you have finished using the symbol, delete it with the LIB$DELETE_SYMBOL routine. If you created a global symbol, make sure you specify the global symbol table in the **tbl-ind** argument. By default, the local symbol table is searched.

3.1.5 Common Blocks

Use common blocks to store data from one image to the next. They are unlikely to be corrupted between the time one image deposits information and another image reads it. Each common block can store 255 bytes of data. The LIB$PUT_COMMON routine writes information to this common block; the LIB$GET_COMMON routine reads information from this common block.

3.1.5.1 How the Process Common Block Is Created

The common block for your process is automatically created for you; no special declaration is necessary. To pass more than 255 bytes of data, put the data in a file and use the common block to pass the name of the file.

3.1.5.2 Modifying or Deleting Data in the Common Block

Data in the process common block cannot be deleted or modified unless LIB$PUT_COMMON is invoked. Therefore, any number of images may be executed between one image and another, provided that LIB$PUT_COMMON has not been invoked. Each subsequent image reads the correct data. Invoking LIB$GET_COMMON to read the common block does not modify the data.

3.1.5.3 Specifying Other Types of Data

Although the descriptions of LIB$PUT_COMMON and LIB$GET_COMMON in the *VMS Run-Time Library Routines Volume* specify a character string for the argument containing the data written to or read from the common block, you can specify other types of data. However, you must pass noncharacter data (as well as character data) by descriptor.

The following program segment reads statistics from the terminal and enters them into a binary file. After all of the statistics are entered into the file, the program places the name of the file into the per-process common block and exits.

```
    .
    .
    .
! Enter statistics
    .
    .
    .
! Put the name of the stats file into common
STATUS = LIB$PUT_COMMON (FILE_NAME (1:LEN))
    .
    .
    .
```

The following program segment reads the file name from the per-process common block and compiles a report using the statistics from that file.

```
    .
    .
    .
! Read the name of the stats file from common
STATUS = LIB$GET_COMMON (FILE_NAME,
2                        LEN)

! Compile the report
    .
    .
    .
```

3.2 Interprocess Communication

Use the following techniques for communicating from one process to another:

- Common event flags
- Global sections
- Mailboxes
- Resource locks
- Shared files

While common event flags and resource locks establish communication, they are most useful for synchronizing events. Therefore, they are covered in Chapter 4. Global sections and shared files are best used for sharing data; therefore, they are covered in Chapter 5. This section describes the use of mailboxes.

3.2.1 Mailboxes

A mailbox is a virtual device used for communication among processes. You must call VMS RMS services or I/O system services to peform actual data transfer.

3.2.1.1 Creating a Mailbox

To create a mailbox, use the SYS$CREMBX system service. SYS$CREMBX creates the mailbox and returns the number of the I/O channel assigned to the mailbox.

The format for the SYS$CREMBX system service is as follows:

SYS$CREMBX ([prmflg], chan, [maxmsg], [bufquo], [promsk], [acmode], [lognam])

When you invoke SYS$CREMBX, you usually specify the following two arguments:

Specifying the I/O Channel

Specify a variable to receive the I/O channel number using the **chan** argument. This argument is required.

Defining a Logical Name for the Mailbox

Specify the logical name to be associated with the mailbox using the **lognam** argument. The logical name identifies the mailbox for other processes and for input/output statements.

The SYS$CREMBX system service also allows you to specify the message size, buffer size, mailbox protection code, and access mode of the mailbox; however, the default values for these arguments are usually sufficient. For more information on SYS$CREMBX, refer to the *VMS System Services Reference Manual*.

3.2.1.2 Temporary and Permanent Mailboxes

By default, a mailbox is deleted when no I/O channel is assigned to it. Such a mailbox is called a temporary mailbox. If you have PRMMBX privilege, you can create a permanent mailbox (specify the **prmflg** argument as 1 when you invoke SYS$CREMBX). A permanent mailbox is not deleted until it is marked for deletion with the SYS$DELMBX system service (requires PRMMBX). Once a permanent mailbox is marked for deletion, it is like a temporary mailbox; when the last I/O channel to the mailbox is deassigned, the mailbox is deleted.

The following statement creates a mailbox named MAIL_BOX. The I/O channel assigned to the mailbox is returned in MBX_CHAN.

```
! I/O channel
INTEGER*2 MBX_CHAN

! Mailbox name
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')

STATUS = SYS$CREMBX (,
2                    MBX_CHAN,  ! I/O channel
2                    ,,,,
2                    MBX_NAME)  ! Mailbox name
```

Note: Do not use MAIL as the logical name for a mailbox or the system will not execute the proper image in response to the DCL command MAIL.

The following program segment creates a permanent mailbox, then creates a subprocess that marks that mailbox for deletion:

```
INTEGER STATUS,
2       SYS$CREMBX
INTEGER*2 MBX_CHAN

! Create permanent mailbox
STATUS = SYS$CREMBX (%VAL(1),      ! Permanence flag
2                   MBX_CHAN,     ! Channel
2                   ,,,,
2                   'MAIL_BOX')   ! Logical name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Create subprocess to delete it
STATUS = LIB$SPAWN ('RUN DELETE_MBX')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

The following program segment executes in the subprocess. Notice that the subprocess must assign a channel to the mailbox and then use that channel to delete the mailbox. Any process that deletes a permanent mailbox, unless it is the creating process, must use this technique. (Use SYS$ASSIGN to assign the channel to the mailbox to ensure that the mailbox already exists. SYS$CREMBX system service assigns a channel to a mailbox; however, SYS$CREMBX also creates the mailbox if it does not already exist.)

```
INTEGER STATUS,
2       SYS$DELMBX,
2       SYS$ASSIGN
INTEGER*2 MBX_CHAN

! Assign channel to mailbox
STATUS = SYS$ASSIGN ('MAIL_BOX',
2                   MBX_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Delete the mailbox
STATUS = SYS$DELMBX (%VAL(MBX_CHAN))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

3.2.1.3 Reading and Writing Data to a Mailbox

The following list describes the three ways you can read and write to a mailbox:

- Synchronous—Read or write to a mailbox and then wait for the cooperating image to perform the opposite operation. Use I/O statements for your programming language. This is the recommended method of addressing a mailbox.
- Immediate—Queue read or write to a mailbox and continue program execution after the operation completes. Use the SYS$QIOW system service.
- Asynchronous—Queue a read or write request to a mailbox and continue program execution while the request executes. Use the SYS$QIO system service. When the read or write operation completes, the I/O status block (if specified) is filled, the event flag (if specified) is set, and the AST routine (if specified) is executed.

Chapter 7 describes the SYS$QIO and SYS$QIOW system services. See *VMS System Services Reference Manual* for more information. DIGITAL recommends that you supply the optional I/O status block parameter when you use these two system services. The contents of the status block varies depending on the QIO function code; refer to the function code descriptions in the *VMS I/O User's Reference Volume* for a description of the appropriate status block.

3.2.1.4 Synchronous Mailbox I/O

Use synchronous I/O when you read or write information to another image and cannot continue until that image responds.

The program segment shown in Example 3–2 opens a mailbox for the first time. To open a mailbox for VAX FORTRAN I/O, use the OPEN statement with the following specifiers: UNIT, FILE, CARRIAGECONTROL, and STATUS. The value for the keyword FILE should be the logical name of a mailbox (SYS$CREMBX allows you to associate a logical name with a mailbox when the mailbox is created). The value for the keyword CARRIAGECONTROL should be 'LIST'. The value for the keyword STATUS should be 'NEW' for the first OPEN statement and 'OLD' for subsequent OPEN statements.

Example 3–2 Opening a Mailbox

```
! Status variable and values
INTEGER STATUS

! Logical unit and name for mailbox
INTEGER MBX_LUN
CHARACTER(*) MBX_NAME
PARAMETER (MBX_NAME = MAIL_BOX)
! Create mailbox
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,  ! Channel
2                    ,,,,
2                    MBX_NAME)  ! Logical name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Get logical unit for mailbox and open mailbox
STATUS = LIB$GET_LUN (MBX_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = MBX_LUN,
2     FILE = MBX_NAME,
2     CARRIAGECONTROL = 'LIST',
2     STATUS = 'NEW')
```

In Example 3-3, one image passes device names to a second image. The second image returns the process name and the terminal associated with the process that allocated each device. A WRITE statement in the first image does not complete until the cooperating process issues a READ statement. (The variable declarations are not shown in the second program because they are very similar to those in the first program.)

Example 3–3 Synchronous I/O Using a Mailbox

```
! DEVICE.FOR

PROGRAM PROCESS_DEVICE

! Status variable
INTEGER STATUS

! Name and I/O channel for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! Logical unit number for FORTRAN I/O
INTEGER MBX_LUN
! Character string format
CHARACTER*(*) CHAR_FMT
PARAMETER (CHAR_FMT = '(A50)')
! Mailbox message
CHARACTER*50 MBX_MESSAGE
   .
   .
   .
! Create the mailbox
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,    ! Channel
2                    ,,,,
2                    MBX_NAME)    ! Logical name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get logical unit for mailbox and open mailbox
STATUS = LIB$GET_LUN (MBX_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = MBX_LUN,
2     FILE = MBX_NAME,
2     CARRIAGECONTROL = 'LIST',
2     STATUS = 'NEW')
! Create subprocess to execute GETDEVINF.EXE
STATUS = SYS$CREPRC (,
2                    'GETDEVINF',   ! Image
2                    ,,,,,
2                    'GET_DEVICE',  ! Process name
2                    %VAL(4),,,)    ! Priority
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Pass device names to GETDEFINF
WRITE (UNIT=MBX_LUN,
2      FMT=CHAR_FMT) 'SYS$DRIVE0'
! Read device information from GETDEFINF
READ (UNIT=MBX_LUN,
2     FMT=CHAR_FMT) MBX_MESSAGE
   .
   .
   .
END
```

Example 3–3 Cont'd. on next page

Example 3–3 (Cont.) Synchronous I/O Using a Mailbox

GETDEVINF.FOR

```
    .
    .
    .
! Create mailbox
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,   ! I/O channel
2                    ,,,,
2                    MBX_NAME)   ! Mailbox name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get logical unit for mailbox and open mailbox
STATUS = LIB$GET_LUN (MBX_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT=MBX_LUN,
2     FILE=MBX_NAME,
2     CARRIAGECONTROL='LIST',
2     STATUS = 'OLD')
! Read device names from mailbox
READ (UNIT=MBX_LUN,
2     FMT=CHAR_FMT) MBX_MESSAGE
! Use SYS$GETJPI to find process and terminal
! Process name:  PROC_NAME (1:P_LEN)
! Terminal name: TERM (1:T_LEN)
    .
    .
    .
MBX_MESSAGE = MBX_MESSAGE//' '//
2             PROC_NAME(1:P_LEN)//' '//
2             TERM(1:T_LEN)
! Write device information to DEVICE
WRITE (UNIT=MBX_LUN,
2      FMT=CHAR_FMT) MBX_MESSAGE
END
```

3.2.1.5 Immediate Mailbox I/O

Use immediate I/O to read or write to another image without waiting for a response from that image. To ensure that the other process receives the information that you write, either (1) do not exit until the other process has a channel to the mailbox or (2) use a permanent mailbox.

Queueing an Immmediate I/O Request

To queue an immediate I/O request, invoke the SYS$QIOW system service. See *VMS System Services Reference Manual* for more information.

Reading Data from the Mailbox

Since immediate I/O is asynchronous, it is possible that a mailbox may contain more than one message or no message when it is read. If the mailbox contains more than one message, the read operation retrieves the messages one at a time in the order in which they were written. If the mailbox contains no message, the read operation generates an end-of-file error.

To allow a cooperating program to differentiate between an empty mailbox and the end of the data being transferred, the process writing the messages should use the IO$_WRITEOF function code to write an end-of-file message to the mailbox as the last piece of data. When the cooperating program reads an empty mailbox, the end-of-file message is returned and the second longword of the I/O status block is 0. When the cooperating program reads an end-of-file message explicitly written to the mailbox, the end-of-file message is returned and the second longword of the I/O status block contains the process identification number of the process that wrote the message to the mailbox.

In Example 3–4, the first program creates a mailbox named MAIL_BOX, writes data to it, and then indicates the end of the data by writing an end-of-file message. The second program creates a mailbox with the same logical name, reads the messages from the mailbox into an array, and stops the read operations when a read operation generates an end-of-file message and the second longword of the I/O status block is nonzero, confirming that the writing process sent the end-of-file message. The processes use common event flag 64 to ensure that SEND.FOR does not exit until RECEIVE.FOR has established a channel to the mailbox. (If RECEIVE.FOR executes first, an error occurs because SYS$ASSIGN cannot find the mailbox.)

Example 3–4 Immediate I/O Using a Mailbox

```
!SEND.FOR
     .
     .
     .
INTEGER*4 STATUS

! Name and channel number for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! Mailbox message
CHARACTER*80 MBX_MESSAGE
INTEGER LEN
CHARACTER*80 MESSAGES (255)
INTEGER MESSAGE_LEN (255)
INTEGER MAX_MESSAGE
PARAMETER (MAX_MESSAGE = 255)
! I/O function codes and status block
INCLUDE '($IODEF)'
INTEGER*4 WRITE_CODE
STRUCTURE /STATUS_BLOCK/
 INTEGER*2 IOSTAT,
2          MSG_LEN
 INTEGER*4 READER_PID
END STRUCTURE
RECORD /STATUS_BLOCK/ IOSTATUS
! System routines
INTEGER SYS$CREMBX,
2       SYS$ASCEFC,
2       SYS$WAITFR,
2       SYS$QIOW
```

Example 3–4 Cont'd. on next page

Example 3–4 (Cont.) Immediate I/O Using a Mailbox

```
! Create the mailbox
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,
2                    ,,,,
2                    MBX_NAME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Fill MESSAGES array
   .
   .
   .
! Write the messages
DO I = 1, MAX_MESSAGE
  WRITE_CODE = IO$_WRITEVBLK .OR. IO$M_NOW
  MBX_MESSAGE = MESSAGES(I)
  LEN = MESSAGE_LEN(I)
  STATUS = SYS$QIOW (,
2                    %VAL(MBX_CHAN),      ! Channel
2                    %VAL(WRITE_CODE),    ! I/O code
2                    IOSTATUS,            ! Status block
2                    ,,
2                    %REF(MBX_MESSAGE),   ! P1
2                    %VAL(LEN),,,,)       ! P2
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  IF (.NOT. IOSTATUS.IOSTAT)
2      CALL LIB$SIGNAL (%VAL(IOSTATUS.STATUS))
END DO
! Write end-of-file
WRITE_CODE = IO$_WRITEOF .OR. IO$M_NOW
STATUS = SYS$QIOW (,
2                  %VAL(MBX_CHAN),      ! Channel
2                  %VAL(WRITE_CODE),    ! End-of-file code
2                  IOSTATUS,            ! Status block
2                  ,,,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. IOSTATUS.IOSTAT)
2    CALL LIB$SIGNAL (%VAL(IOSTATUS.IOSTAT))
   .
   .
   .
! Make sure cooperating process can read the information
! by waiting for it to assign a channel to the mailbox
STATUS = SYS$ASCEFC (%VAL(64),
2                    'CLUSTER',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$WAITFR (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

Example 3–4 Cont'd. on next page

Example 3–4 (Cont.) Immediate I/O Using a Mailbox

RECEIVE.FOR

```
INTEGER STATUS

INCLUDE '($IODEF)'
INCLUDE '($SSDEF)'

! Name and channel number for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! QIO function code
INTEGER READ_CODE
! Mailbox message
CHARACTER*80 MBX_MESSAGE
INTEGER*4    LEN
! Message arrays
CHARACTER*80 MESSAGES (255)
INTEGER*4    MESSAGE_LEN (255)
! I/O status block
STRUCTURE /STATUS_BLOCK/
 INTEGER*2 IOSTAT,
2          MSG_LEN
 INTEGER*4 READER_PID
END STRUCTURE
RECORD /STATUS_BLOCK/ IOSTATUS
! System routines
INTEGER SYS$ASSIGN,
2       SYS$ASCEFC,
2       SYS$SETEF,
2       SYS$QIOW
! Create the mailbox and let the other process know
STATUS = SYS$ASSIGN (MBX_NAME,
2                    MBX_CHAN,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$ASCEFC (%VAL(64),
2                    'CLUSTER',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$SETEF (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Example 3–4 Cont'd. on next page

Example 3–4 (Cont.) Immediate I/O Using a Mailbox

```
! Read first message
READ_CODE = IO$_READVBLK .OR. IO$M_NOW
LEN = 80
STATUS = SYS$QIOW (,
2                  %VAL(MBX_CHAN),      ! Channel
2                  %VAL(READ_CODE),     ! Function code
2                  IOSTATUS,            ! Status block
2                  ,,
2                  %REF(MBX_MESSAGE),   ! P1
2                  %VAL(LEN),,,,)       ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF ((.NOT. IOSTATUS.IOSTAT) .AND.
2  (IOSTATUS.IOSTAT .NE. SS$_ENDOFFILE)) THEN
  CALL LIB$SIGNAL (%VAL(IOSTATUS.IOSTAT))
ELSE IF (IOSTATUS.IOSTAT .NE. SS$_ENDOFFILE) THEN
  I = 1
  MESSAGES(I) = MBX_MESSAGE
  MESSAGE_LEN(I) = IOSTATUS.MSG_LEN
END IF
! Read messages until cooperating process writes end-of-file
DO WHILE (.NOT. ((IOSTATUS.IOSTAT .EQ. SS$_ENDOFFILE) .AND.
2                (IOSTATUS.READER_PID .NE. 0)))

  STATUS = SYS$QIOW (,
2                  %VAL(MBX_CHAN),      ! Channel
2                  %VAL(READ_CODE),     ! Function code
2                  IOSTATUS,            ! Status block
2                  ,,
2                  %REF(MBX_MESSAGE),   ! P1
2                  %VAL(LEN),,,,)       ! P2
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  IF ((.NOT. IOSTATUS.IOSTAT) .AND.
2     (IOSTATUS.IOSTAT .NE. SS$_ENDOFFILE)) THEN
    CALL LIB$SIGNAL (%VAL(IOSTATUS.IOSTAT))
  ELSE IF (IOSTATUS.IOSTAT .NE. SS$_ENDOFFILE) THEN
    I = I + 1
    MESSAGES(I) = MBX_MESSAGE
    MESSAGE_LEN(I) = IOSTATUS.MSG_LEN
  END IF

END DO
   .
   .
   .
```

3.2.1.6 Asynchronous Mailbox I/O

Use asynchronous I/O to queue a read or write request to a mailbox. To ensure that the other process receives the information you write, either (1) exit *after* the other process has a channel to the mailbox or (2) use a permanent mailbox.

To queue an asynchronous I/O request, invoke the SYS$QIO system service as shown in Section 3.2.1.5; however, when specifying the function codes, do not specify the IO$M_NOW modifier. The SYS$QIO system service allows you to specify an AST to be executed or an event flag to be set when the I/O operation completes.

Example 3–5 calculates gross income and taxes and then uses the results to calculate net income. INCOME.FOR uses SYS$CREPRC, specifying a termination mailbox, to create a subprocess to calculate taxes (CALC_TAXES) while INCOME calculates gross income. INCOME issues an asynchronous read to the termination mailbox specifying an event flag to be set when the read completes. (The read completes when CALC_TAXES completes terminating the created process and causing the system to write to the termination mailbox.) After finishing its own gross income calculations, INCOME.FOR waits for the flag that indicates CALC_TAXES has completed and then figures net income.

CALC_TAXES.FOR passes the tax information to INCOME.FOR using the installed common block created from INSTALLED.FOR (Section 5.3.4.1 describes installed common blocks).

Example 3–5 Asynchronous I/O Using a Mailbox

```
!INSTALLED.FOR

! Installed common block to be linked with INCOME.FOR and
! CALC_TAXES.FOR.
! Unless the shareable image created from this file is
! in SYS$SHARE, you must define a group logical name
! INSTALLED and equivalence it to the full file specification
! of the shareable image.
INTEGER*4 INCOME (200),
2         TAXES (200),
2         NET (200)
COMMON /CALC/ INCOME,
2             TAXES,
2             NET

END
!INCOME.FOR
! Status and system routines
    .
    .
    .
INCLUDE '($SSDEF)'
INCLUDE '($IODEF)'
INTEGER STATUS,
2       LIB$GET_LUN,
2       LIB$GET_EF,
2       SYS$CLREF,
2       SYS$CREMBX,
2       SYS$CREPRC,
2       SYS$GETDVIW,
2       SYS$QIO,
2       SYS$WAITFR
```

Example 3–5 Cont'd. on next page

Example 3–5 (Cont.) Asynchronous I/O Using a Mailbox

```
! Set up for SYS$GETDVI
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Declare itmlst
RECORD /ITMLST/ DVILIST (2)
INTEGER*4 UNIT_BUF,
2         UNIT_LEN
EXTERNAL DVI$_UNIT
! Name and I/O channel for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
INTEGER*4 MBX_LUN           ! Logical unit number for I/O
CHARACTER*84 MBX_MESSAGE    ! Mailbox message
INTEGER*4 READ_CODE,
2         LENGTH
! I/O status block
STRUCTURE /STATUS_BLOCK/
 INTEGER*2 IOSTAT,
2          MSG_LEN
 INTEGER*4 READER_PID
END STRUCTURE
RECORD /STATUS_BLOCK/ IOSTATUS
! Declare calculation variables in installed common
INTEGER*4 INCOME (200),
2         TAXES (200),
2         NET (200)
COMMON /CALC/ INCOME,
2             TAXES,
2             NET
! Flag to indicate taxes calculated
INTEGER*4 TAX_DONE
! Get and clear an event flag
STATUS = LIB$GET_EF (TAX_DONE)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(TAX_DONE))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Create the mailbox
STATUS = SYS$CREMBX (,
2                   MBX_CHAN,
2                   ,,,,
2                   MBX_NAME)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
```

Example 3–5 Cont'd. on next page

Example 3–5 (Cont.) Asynchronous I/O Using a Mailbox

```
! Get unit number of the mailbox
DVILIST(1).BUFLEN    = 4
DVILIST(1).CODE      = %LOC(DVI$_UNIT)
DVILIST(1).BUFADR    = %LOC(UNIT_BUF)
DVILIST(1).RETLENADR = %LOC(UNIT_LEN)
DVILIST(2).END_LIST  = 0
STATUS = SYS$GETDVIW (,
2                      %VAL(MBX_CHAN),  ! Channel
2                      MBX_NAME,        ! Device
2                      DVILIST,         ! Item list
2                      ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Create subprocess to calculate taxes
STATUS = SYS$CREPRC (,
2                     'CALC_TAXES', ! Image
2                     ,,,,,
2                     'CALC_TAXES', ! Process name
2                     %VAL(4),      ! Priority
2                     ,
2                     %VAL(UNIT_BUF),)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Asynchronous read to termination mailbox
! sets flag when tax calculations complete
READ_CODE = IO$_READVBLK
LENGTH = 84
STATUS = SYS$QIO (%VAL(TAX_DONE),    ! Indicates read complete
2                 %VAL(MBX_CHAN),    ! Channel
2                 %VAL(READ_CODE),   ! Function code
2                 IOSTATUS,,,        ! Status block
2                 %REF(MBX_MESSAGE),! P1
2                 %VAL(LENGTH),,,,) ! P2
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Calculate incomes
   .
   .
   .
! Wait until taxes are calculated
STATUS = SYS$WAITFR (%VAL(TAX_DONE))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Check mailbox I/O
IF (.NOT. IOSTATUS.IOSTAT)
2    CALL LIB$SIGNAL (%VAL(IOSTATUS.IOSTAT))
! Calculate net income after taxes
   .
   .
   .
END
```

Example 3–5 Cont'd. on next page

Example 3–5 (Cont.) Asynchronous I/O Using a Mailbox

CALC_TAXES.FOR

```
! Declare calculation variables in installed common
INTEGER*4 INCOME (200),
2         TAXES (200),
2         NET (200)
COMMON /CALC/ INCOME,
2             TAXES,
2             NET

! Calculate taxes
    .
    .
    .
END
```

3.3 System Information

The VMS operating system provides services for communicating with the system to collect information.

3.3.1 Timer Statistics

You can collect the following timer statistics from the system:

- Elapsed time—Actual time that has passed since setting a timer
- CPU time—CPU time that has passed since setting a timer
- Buffered I/O—Number of buffered I/O operations that have occurred since setting a timer
- Direct I/O—Number of direct I/O operations that have occurred since setting a timer
- Page faults—Number of page faults that have occurred since setting a timer

You obtain these statistics by invoking the following routines:

- LIB$INIT_TIMER—Allocates and initializes space for collecting the statistics. You should specify the **handle-adr** argument as a variable with a value of 0 to ensure the modularity of your program. When you specify the argument, the system collects the information in a specially allocated area in dynamic storage. Then, if a program unit calling your program unit also sets a timer, your program timer statistic remains intact.
- LIB$SHOW_TIMER—Obtains one or all of five statistics (elaspsed time, CPU time, buffered I/O, direct I/O, and page faults); the statistics are formatted for output. The **handle-adr** argument must be the same value as specified for LIB$INIT_TIMER (do not modify this variable). Specify the **code** argument to obtain one particular statistic rather than all the statistics.

You can let the system write the statistics to SYS$OUTPUT (the default) or you can process the statistics with a subprogram of your own. To process the statistics yourself, specify the name of your subprogram in the **action-rtn** argument. You can pass one argument to your subprogram by naming it in the **user-arg** argument. If you use your own subprogram, it must be written as an integer function and return an error code (return a value of 1 for success). This error code becomes the error code returned by LIB$SHOW_TIMER. Two arguments are passed to your function: the first is a passed-length character string containing the formatted statistics, and the second is the value of the fourth argument (if any) specified to LIB$SHOW_TIMER.

- LIB$STAT_TIMER—Obtains one of five unformatted statistics. Specify the statistic you want in the **code** argument. Specify a storage area for the statistic in **value**. The **handle-adr** argument must be the same value as you specified for LIB$INIT_TIMER.
- LIB$FREE_TIMER—You should invoke this procedure when you are done with the timer to ensure the modularity of your program. The value in the **handle-adr** argument must be the same as specified for LIB$INIT_TIMER.

You must invoke LIB$INIT_TIMER to allocate storage for the timer. You should invoke LIB$FREE_TIMER before you exit from your program unit. In between, you can invoke LIB$SHOW_TIMER or LIB$STAT_TIMER, or both, as often as you want. Example 3–6 invokes LIB$SHOW_TIMER and uses a user-written subprogram to either display the statistics or write them to a file.

Example 3–6 Displaying and Writing Timer Statistics

```
      .
      .
      .
! Timer arguments
INTEGER*4 TIMER_ADDR,
2         TIMER_DATA,
2         TIMER_ROUTINE
EXTERNAL  TIMER_ROUTINE
! Declare library procedures as functions
INTEGER*4 LIB$INIT_TIMER,
2         LIB$SHOW_TIMER
EXTERNAL  LIB$INIT_TIMER,
2         LIB$SHOW_TIMER
! Work variables
CHARACTER*5 REQUEST
INTEGER*4   STATUS
! User request - either WRITE or FILE
INTEGER*4   WRITE,
2           FILE
PARAMETER  (WRITE = 1,
2           FILE = 2)
! Get user request
WRITE (UNIT=*, FMT='($,A)') ' Request: '
ACCEPT *, REQUEST
IF (REQUEST .EQ. 'WRITE') TIMER_DATA = WRITE
IF (REQUEST .EQ. 'FILE') TIMER_DATA = FILE
```

Example 3–6 Cont'd. on next page

Example 3–6 (Cont.) Displaying and Writing Timer Statistics

```
! Set timer
STATUS = LIB$INIT_TIMER (TIMER_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
! Get statistics
STATUS = LIB$SHOW_TIMER (TIMER_ADDR,,
2                         TIMER_ROUTINE,
2                         TIMER_DATA)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
! Free timer
STATUS = LIB$FREE_TIMER (TIMER_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
INTEGER FUNCTION TIMER_ROUTINE (STATS,
2                                TIMER_DATA)
! Dummy arguments
CHARACTER*(*) STATS
INTEGER TIMER_DATA
! Logical unit number for file
INTEGER STATS_FILE
! User request
INTEGER WRITE,
2       FILE
PARAMETER (WRITE = 1,
2          FILE = 2)
! Return code
INTEGER SUCCESS,
2       FAILURE
PARAMETER (SUCCESS = 1,
2          FAILURE = 0)
! Set return status to success
TIMER_ROUTINE = SUCCESS
! Write statistics or file them in STATS.DAT
IF (TIMER_DATA .EQ. WRITE) THEN
  TYPE *, STATS
ELSE IF (TIMER_DATA .EQ. FILE) THEN
  CALL LIB$GET_LUN (STATS_FILE)
  OPEN (UNIT=STATS_FILE,
2       FILE='STATS.DAT')
  WRITE (UNIT=STATS_FILE,
2        FMT='(A)') STATS
ELSE
  TIMER_ROUTINE = FAILURE
END IF
END
```

You can use the system service SYS$GETSYI to obtain more detailed system information on boot time, the cluster, processor type, emulated instructions, nodes, paging files, swapping files, and hardware and software versions. With SYS$GETQUI and LIB$GETQUI, you can obtain queue information.

3.3.2 System Time

The VMS operating system recognizes two types of time, as follows:

- Absolute time—A specified date or time of day, or both
- Delta time—A number of days or units of time within a day, or both

3.3.2.1 Absolute Time Format

The VMS operating system uses the following format for absolute time. The full date and time require a character string of 23 characters. The punctuation is required.

dd-mmm-yyyy hh:mm:ss.ss

dd	Day of the month (two characters)
mmm	First three letters of the month in uppercase (three characters)
yyyy	Year (four characters)
hh	Hour of the day in 24-hour format (two characters)
mm	Minute (two characters)
ss.ss	Second and hundredths of a second (five characters)

3.3.2.2 Delta Time Format

The VMS operating system uses the following format for delta time. The full date and time require a character string of 16 characters. The punctuation is required.

dddd hh:mm:ss.ss

dddd	Days (four characters)
hh	Hours (two characters)
mm	Minutes (two characters)
ss.ss	Seconds and hundredths of seconds (five characters)

Internally, the system maintains absolute time as an integer value representing the number of 100-nanosecond units since midnight on 17-NOV-1858 (the base date for the system). A delta time is maintained as an integer value representing an amount of time in 100-nanosecond units. The absolute time is maintained as a positive number and the delta time as a negative number, in quadwords.

3.3.2.3 Current Time

The LIB$DATE_TIME routine returns a character string containing the current date and time in absolute time format. The full string requires a declaration of 23 characters. If you specify a shorter string, the value is truncated. A declaration of 16 characters obtains only the date. The following example displays the current date and time:

```
! Formatted date and time
CHARACTER*23 DATETIME
! Status and library procedures
INTEGER*4 STATUS,
2         LIB$DATE_TIME
EXTERNAL  LIB$DATE_TIME
STATUS = LIB$DATE_TIME (DATETIME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
TYPE *, DATETIME
```

You can obtain the current date and time in internal format with the SYS$GETTIM system service. You can convert from internal to character format with the SYS$ASCTIM system service or a directive to the SYS$FAO system service and convert back to internal format with the SYS$BINTIM system service.

3.3.2.4 Time Manipulation

Use the following general procedures for manipulating times (for example, finding the delta difference between two absolute times, adding a delta time to an absolute time, or subtracting a delta time from an absolute time):

- Convert to internal format—Obtain the time in, or convert the time to, internal format. Use SYS$GETTIM to get the current time in internal format or SYS$BINTIM to convert a formatted time to an internal time. You can also use LIB$DATE_TIME to obtain the time, LIB$CVT_FROM_INTERNAL_TIME to convert an internal time to an external time, and LIB$CVT_TO_INTERNAL to convert from an external time to an internal time.
- Manipulate the times—Add, subtract, or otherwise manipulate the times. Use the LIB$ADDX and LIB$SUBX routines to add and subtract times, since the times are defined in integer arrays. Use LIB$ADD_TIMES and LIB$SUB_TIMES to add and subtract two quadword times. When manipulating delta times, remember that they are stored as negative numbers. For example, to add a delta time to an absolute time, you must subtract the delta time from the absolute time. Use LIB$MULT_DELTA_TIME and LIB$MULTF_DELTA_TIME to multiply delta times by scalar and floating scalar.
- Format the times—Format the result, as desired, with SYS$BINTIM or SYS$FAO. You can also use LIB$FORMAT_DATE_TIME.

Example 3–7 calculates the difference between the current time and a time input in absolute format and then displays the result as a delta time. If the input time is later than the current time, the difference is a negative value (delta time) and can be displayed directly. If the input time is an earlier time, the difference is a positive value (absolute time) and must be converted to a delta time before being displayed. To change an absolute time to a delta time, negate the time array by subtracting it from 0 (specified as an integer array) using the LIB$SUBX routine. For the absolute or delta time format, see Section 3.3.2.

Example 3–7 Calculating and Displaying the Time

```
   .
   .
   .
! Internal times
! Input time in absolute format, dd-mmm-yyyy hh:mm:ss.ss
!
INTEGER*4 CURRENT_TIME (2),
2         PAST_TIME (2),
2         TIME_DIFFERENCE (2),
2         ZERO (2)
DATA ZERO /0,0/
! Formatted times
CHARACTER*23 PAST_TIME_F
CHARACTER*16 TIME_DIFFERENCE_F
! Status
INTEGER*4 STATUS
! Integer functions
INTEGER*4 SYS$GETTIM,
2         LIB$GET_INPUT,
2         SYS$BINTIM,
2         LIB$SUBX,
2         SYS$ASCTIM
! Get current time
STATUS = SYS$GETTIM (CURRENT_TIME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get past time and convert to internal format
STATUS = LIB$GET_INPUT (PAST_TIME_F,
2                       'Past time (in absolute format): ')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SYS$BINTIM (PAST_TIME_F,
2                    PAST_TIME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Subtract past time from current time
STATUS = LIB$SUBX (CURRENT_TIME,
2                  PAST_TIME,
2                  TIME_DIFFERENCE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! If resultant time is in absolute format (positive value means
! most significant bit is not set), convert it to delta time
IF (.NOT. (BTEST (TIME_DIFFERENCE(2),31))) THEN
  STATUS = LIB$SUBX (ZERO,
2                    TIME_DIFFERENCE,
2                    TIME_DIFFERENCE)
END IF
! Format time difference and display
STATUS = SYS$ASCTIM (, TIME_DIFFERENCE_F,
2                    TIME_DIFFERENCE,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
TYPE *, 'Time difference = ', TIME_DIFFERENCE_F
END
```

If you are ignoring the time portion of date/time (that is, working just at the date level), the LIB$DAY routine might simplify your calculations. LIB$DAY returns to you the number of days from the base system date to a given date.

3.4 Intersystem Communication

To communicate between images on different systems, perform the following operations:

- Request the network connection (initiating process)
- Complete the network connection (remote process)
- Exchange messages (both processes)
- Terminate the network connection (process that receives the final message)

3.4.1 Requesting a Network Connection

To request a network connection, open a file that uses a network task specification of the following format:

node"access-control-string"::"TASK=command-procedure"

- Node—Specifies the node name of the remote system.
- "Access-control-string"—Specifies the user name and associated password of an account on the remote system. The remote system uses the access control string to ensure that you have valid access rights to the system. (This string may be omitted if the calling process has a proxy account on the remote node. For more information, see the description of the Authorize Utility in the *VMS Authorize Utility Manual.*)
- "TASK=command-procedure"—Specifies the task to be executed on the remote node. The command procedure, which must invoke the program that completes the network connection, is a user-written command procedure that must be in the default directory (on the default disk) of the account named in the access control string. (The login command procedure of the remote account is executed before the system searches for the command procedure; therefore, if the login command procedure changes the default device or directory, the command procedure must be in that device and directory rather than the SYS$LOGIN device and directory.)

The following program segment requests a network connection to the remote system PHILLY. To prevent a security problem, the program constructs the access control string by prompting the user for a user name and password. (To prevent the password from being echoed as the user types it, use the SYS$QIO system service and the IO$M_NOECHO modifier, as described in Section 7.5.4.)

```
! Status variable
INTEGER STATUS

! Logical unit for network connection
INTEGER NET_LUN

! User name and password
CHARACTER*15 USERNAME,
2            PASSWORD
INTEGER USERNAME_LEN,
2       PASSWORD_LEN
! Task specification string
CHARACTER*80 TASK
```

```
! Declare system routines
INTEGER LIB$GET_LUN,
2       LIB$GET_INPUT
! Get logical unit for network connection
STATUS = LIB$GET_LUN (NET_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get user name and password
STATUS = LIB$GET_INPUT (USERNAME,
2                       'USERNAME: ',
2                       USERNAME_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (PASSWORD,
2                       'PASSWORD: ',
2                       PASSWORD_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Create a network access string of the form:
! PHILLY"username password"::"TASK=BUDGET"
TASK = 'PHILLY"'//
2      USERNAME(1:USERNAME_LEN)//' '//
2      PASSWORD(1:PASSWORD_LEN)//
2      '"::"TASK=BUDGET"'
OPEN (UNIT=NET_LUN,
2     FILE = TASK,
2     STATUS = 'OLD')
    .
    .
    .
```

3.4.2 Completing a Network Connection

To complete a network connection, the program that is invoked by the command procedure named in the connection request opens a file with the value SYS$NET. In the following example, the command procedure BUDGET.COM invokes the image NET_IMAGE, which completes the network connection requested in the previous example.

BUDGET.COM

```
$ RUN NET_IMAGE
$ PURGE/KEEP=2 NETSERVER.LOG
```

NET_IMAGE.FOR

```
! Status variable
INTEGER STATUS

! Logical unit number for network connection
INTEGER NET_LUN

! Declare system routines
INTEGER LIB$GET_LUN

! Get a logical unit number and
! complete the network connection
STATUS = LIB$GET_LUN (NET_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

OPEN (UNIT = NET_LUN,
2     FILE = 'SYS$NET',
2     STATUS = 'OLD')
    .
    .
    .
```

The NETSERVER.LOG file, which is purged in the command procedure, is created in the default directory of the remote account if the remote system can be accessed and the account is valid. The NETSERVER.LOG file describes the network transaction regardless of whether the connection completes successfully.

3.4.3 Exchanging Messages

To exchange messages, cooperating programs can use programming language read and write statements. In Example 3–8, GET_STATS.FOR requests a network connection. If SEND_STATS.FOR completes the connection, SEND_STATS.FOR writes the statistics and GET_STATS.FOR reads them. The command procedure SEND_STATS.COM must be in the default directory of the remote account specified by the user executing GET_STATS.FOR.

Example 3–8 Exchanging Messages

```
!GET_STATS.FOR
    .
    .
    .
! Communicates with SEND_STATS on remote node PHILLY.
! User must supply username/password from an account
! on remote system.

! Status variables and values
INTEGER STATUS,
2       IOSTAT,
2       IO_OK
PARAMETER (IO_OK = 0)
INCLUDE '($FORDEF)'
! Logical unit for network connection
INTEGER LUN
! Statistics
INTEGER STATS (2500)
INTEGER MAX_STATS /2500/
! User name and password
CHARACTER*15 USERNAME,
2            PASSWORD
INTEGER USERNAME_LEN,
2       PASSWORD_LEN
! Network task string
CHARACTER*80 TASK
! Declare system routines
INTEGER LIB$GET_LUN,
2       LIB$GET_INPUT
! Get logical unit for network connection
STATUS = LIB$GET_LUN (LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Example 3–8 Cont'd. on next page

Example 3–8 (Cont.) Exchanging Messages

```
! Get user name on remote system
STATUS = LIB$GET_INPUT (USERNAME,
2                       'USERNAME: ',
2                       USERNAME_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get matching password
STATUS = LIB$GET_INPUT (PASSWORD,
2                       'PASSWORD: ',
2                       PASSWORD_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Concatenate node, user name, password, and
! command procedure name to create task name of the
! format:  PHILLY"username password"::"TASK = SEND_STATS"
TASK = 'PHILLY"'//
2      USERNAME(1:USERNAME_LEN)//' '//
2      PASSWORD(1:PASSWORD_LEN)//
2       '"::"TASK=SEND_STATS"'
! Request network connection
OPEN (UNIT=LUN,
2     FILE = TASK,
2     STATUS = 'OLD')
! Read statistics
I = 1

READ (UNIT = LUN,
2     FMT = '(I4)',
2     IOSTAT = IOSTAT) STATS (I)
DO WHILE ((IOSTAT .EQ. IO_OK) .AND. (I .LT. MAX_STATS))
  I = I + 1
  READ (UNIT = LUN,
2       FMT = '(I4)',
2       IOSTAT = IOSTAT) STATS(I)
END DO
! Check that IOSTAT is okay or end of file
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA)
2    CALL LIB$SIGNAL(%VAL(STATUS))
END IF

! Terminate network connection
CLOSE (LUN)

END
```

SEND_STATS.COM

```
$ RUN SEND_STATS
$ PURGE/KEEP=2 NETSERVER.LOG
```

Example 3–8 Cont'd. on next page

Example 3–8 (Cont.) Exchanging Messages

SEND_STATS.FOR

```
! Passes statistics to a remote node.

! Status variable
INTEGER STATUS

! Statistics
INTEGER STATS (2500)
INTEGER MAX_STATS /2500/

! Logical unit number for network connection
INTEGER LUN

! Library routines
INTEGER LIB$GET_LUN

! Get logical unit number
STATUS = LIB$GET_LUN (LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Complete network connection
OPEN (UNIT = LUN,
2     FILE = 'SYS$NET',
2     STATUS = 'OLD')

! Pass statistics to remote node
DO I=1,MAX_STATS
  WRITE (UNIT = LUN,
2        FMT = '(I4)') STATS(I)
END DO

END
```

3.4.4 Terminating a Network Connection

To terminate a network connection, the program must close the file that has been opened for the network connection. To prevent losing data, the program that receives the last message should terminate the network connection. When a program terminates a network connection, the cooperating program receives an end-of-file message on the subsequent read operation.

4 Synchronization

Synchronization techniques vary depending on whether the program units in question are in the same program, in different programs executing in the same process, or in different programs executing in different processes. If your application requires the execution of two or more programs, you can execute the programs sequentially, using one process; sequentially, using multiple processes; or concurrently, using multiple processes. For more information on using processes, refer to Chapter 2.

Program synchronization can be done using the following VMS operating system resources:

- Local and common event flags
- Asynchronous system traps (AST)
- Timers
- Synchronous routines
- Resource locks
- Parallel processing facility (PPL$) routines
- Passing control from one program to another

This chapter describes how and when to use each of these resources for program synchronization.

4.1 Synchronizing Operations with Event Flags

To synchronize events with event flags, a procedure sets an event flag bit when it has completed a section of code. Another procedure examines the value of the event flag bit. If the event flag bit is set, the second procedure can resume execution.

4.1.1 Types of Event Flags

Event flags are divided into four clusters—two for local event flags and two for common event flags. Local event flags are process specific and are used to synchronize events within a program or to pass information from the current image to an image executed later by the same process. Common event flags are group specific; use them to synchronize events among images executing in different processes (provided that the processes are in the same group).

Refer to Table 4–1 for a summary of event flag numbers and usage.

Table 4–1 Event Flags

Cluster Number	Flag Number	Type	Usage
0	0	Local	Default flag used by system routines.
0	1 to 23	Local	May be used in system routines. When an event flag is requested, not returned unless it has been specifically freed previously.
0	24 to 31	Local	Reserved for DIGITAL use only.
1	32 to 63	Local	Available for general use.
2	64 to 95	Common	Available for general use.
3	96 to 127	Common	Available for general use.

4.1.2 General Guidelines for Using Event Flags

To use event flags, follow these general steps:

1. Allocate local event flags or associate common event flags for your use.
2. Set or clear the event flag.
3. Read the event flag.
4. Suspend program execution until an event flag is set.
5. Deallocate the local event flags or dissociate common event flags when no longer needed.

Use system services and run-time library routines to accomplish these event flag tasks. Refer to Table 4–2 for a summary of the event flag routines.

Table 4–2 Event Flag Routines

Routine	Event Flag Task
LIB$GET_EF	Allocate any local event flag
LIB$RSERVE_EF	Allocate a specific local event flag
SYS$ASCEFC	Associate a common event flag cluster
SYS$CLREF	Clear a local or common event flag
SYS$SETEF	Set a local or common event flag
SYS$READEF	Read a local or common event flag
SYS$WAITFR	Wait for a specific local or common event flag to be set
SYS$WFLOR	Wait for one of several local or common event flags to be set
SYS$WFLAND	Wait for several local or common event flags to be set
SYS$SYNCH	Wait for a local or common event flag to be set and for non-zero I/O status block
LIB$FREE_EF	Deallocate a local event flag
SYS$DACEFC	Dissociate a common event flag cluster

4.1.3 Using Local Event Flags

Local event flags are automatically available to each program. They are not automatically initialized. However, if an event flag is passed to a system service such as SYS$GETJPI, the service initializes the flag before using it.

Other system services use event flags to synchronize their work. The following table lists other system services that use event flags to synchronize their work:

Service	Routine
Input/output	SYS$QIO and SYS$QIOW
Process control	SYS$GETJPI, SYS$GETJPIW, SYS$GETSYI, SYS$GETSYIW, SYS$GETDVI, and SYS$GETDVIW
Lock management	SYS$ENQ and SYS$ENQW
Timer and time conversion	SYS$SETIMR

When using local event flags, use the event flag routines as follows:

1. To ensure that the event flag you are using is not accessed and changed by other uses, reserve the event flag by using LIB$GET_EF or LIB$RESERVE_EF. If free, these routines return an event flag number.
2. Before using the event flag, free it using SYS$CLREF, unless you pass the event flag to a routine that clears it for you.
3. When an event that is relevant to other program components is completed, set the event flag with SYS$SETEF.
4. A program component can read the event flag to determine what has happened and act accordingly. Use SYS$READEF.

5 The program components that evaluate event flag status can be placed in a wait state. Then, when the event flag is set, execution is resumed. Use SYS$WAITFR, SYS$WFLOR, SYS$WFLAND, or SYS$SYNCH routines.

6 When the event flag is no longer required, clear it. Use LIB$FREE_EF.

The following example uses LIB$GET_EF to choose a local event flag and then uses SYS$CLREF to set the event flag to zero (clear the event flag). (Note that run-time library routines require an event flag number to be passed by reference and system services require an event flag number to be passed by value.)

```
INTEGER FLAG,
2       STATUS,
2       LIB$GET_EF,
2       SYS$CLREF

STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

4.1.4 Using Common Event Flags

Common event flags are manipulated like local event flags. However, common event flag clusters are not automatically allocated to a program. Before referencing a common event flag, a program must create a common event flag cluster by associating it with a name. Once the name is associated with the cluster, the program can reference any flag in the cluster.

4.1.4.1 Associating a Name with a Common Event Flag Cluster

To associate a name with a common event flag cluster, use the SYS$ASCEFC system service. The first program to name a common event flag cluster creates it; all flags in a newly created cluster are clear. Other processes that have the same UIC group number as the creator of the cluster can reference the cluster by invoking SYS$ASCEFC and specifying the cluster name.

Different processes may associate the same name with different common event flag clusters; as long as the name is the same, the processes reference the same cluster. It is the bit offset within the cluster, rather than the number of the bit, that is used to reference the bit.

4.1.4.2 Temporary Common Event Flag Clusters

By default, a cluster name and common event flag cluster are dissociated when the image that associated them exits. When the last image associated with a cluster is dissociated, the common event flag cluster is deleted. Clusters that are deleted after all images are dissociated are called temporary clusters. You can also use the system service SYS$DACEFC to dissociate a common cluster.

4.1.4.3 Permanent Common Event Flag Clusters

If you have PRMCEB privilege, you can create a permanent common event flag cluster (set the **perm** argument to 1 when you invoke SYS$ASCEFC). A permanent event flag cluster is not deleted until after it is marked for deletion with the SYS$DLCEFC system service (requires PRMCEB). Once a permanent cluster is marked for deletion, it is like a temporary cluster; when the last image associated with the cluster is dissociated, the cluster is deleted.

In the following examples, the first program segment associates common event flag cluster 3 with the name CLUSTER and then clears the second event flag in the cluster. The second program segment associates common event flag cluster 2 with the name CLUSTER, then sets the second event flag in the cluster (the flag cleared by the first program segment).

Example 1

```
STATUS = SYS$ASCEFC (%VAL(96),
2                    'CLUSTER',,)
STATUS = SYS$CLREF (%VAL(98))
```

Example 2

```
STATUS = SYS$ASCEFC (%VAL(64),
2                    'CLUSTER',,)
STATUS = SYS$SETEF (%VAL(66))
```

For clearer code, rather than using a specific event flag number, use one variable to contain the bit offset you need and one variable to contain the number of the first bit in the common event flag cluster that you are using. To reference the common event flag, add the offset to the number of the first bit. The following examples accomplish exactly the same result as the previous examples:

Example 1

```
INTEGER*4 BASE,
2         OFFSET
PARAMETER (BASE = 96)

OFFSET=2
STATUS = SYS$ASCEFC (%VAL(BASE),
2                    'CLUSTER',,)
STATUS = SYS$CLREF (%VAL(BASE+OFFSET))
```

Example 2

```
INTEGER*4 BASE,
2         OFFSET
PARAMETER (BASE = 64)

OFFSET=2
STATUS = SYS$ASCEFC (%VAL(BASE),
2                    'CLUSTER',,)
STATUS = SYS$SETEF (%VAL(BASE+OFFSET))
```

Common event flags are often used for communicating between a parent process and a created subprocess. The following parent process associates the name CLUSTER with a common event flag cluster, creates a subprocess, and then waits for the subprocess to set event flag 64:

```
INTEGER*4 BASE,
2         OFFSET
PARAMETER (BASE   = 64,
2          OFFSET = 0)
    .
    .
    .
! Associate common event flag cluster with name
STATUS = SYS$ASCEFC (%VAL(BASE),
2                    'CLUSTER',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Create subprocess to execute concurrently
MASK = IBSET (MASK,0)
STATUS = LIB$SPAWN ('RUN REPORTSUB', ! Image
2                   'INPUT.DAT',     ! SYS$INPUT
2                   'OUTPUT.DAT',    ! SYS$OUTPUT
2                   MASK)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Wait for response from subprocess
STATUS = SYS$WAITFR (%VAL(BASE+OFFSET))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
    .
    .
    .
```

REPORTSUB, the program executing in the subprocess, associates the name CLUSTER with a common event flag cluster, performs some set of operations, sets event flag 64 (allowing the parent to continue execution), and continues executing.

```
INTEGER*4 BASE,
2         OFFSET
PARAMETER (BASE   = 64,
2          OFFSET = 0)
    .
    .
    .
                     ! Do operations necessary for
                     ! continuation of parent process
    .
    .
    .
! Associate common event flag cluster with name
STATUS = SYS$ASCEFC (%VAL(BASE),
2                    'CLUSTER',,)
IF (.NOT. STATUS)
2  CALL LIB$SIGNAL (%VAL(STATUS))

! Set flag for parent process to resume
STATUS = SYS$SETEF (%VAL(BASE+OFFSET))
    .
    .
    .
```

4.2 Using Asynchronous System Traps

Asynchronous system traps (AST) are interrupts that occur asynchronously to a program's execution. You can use them to signal a program to execute a routine whenever a certain condition occurs.

The routine executed upon delivery of an AST is called an AST routine. It is coded and referenced like any other subroutine. The differences are that it is executed only after an AST is received by the program and is called asynchronously by the operating system, not by the current image.

When the AST routine is finished, the program that was interrupted resumes execution from the point of interruption.

To deliver an AST, you use system services that specify the address of the AST routine. Then, the system delivers the AST (that is, transfers control to your subprogram) at a particular time or in response to a particular event.

The AST routine must observe the following restrictions:

- Arguments—The queuing mechanism for an AST does not provide for returning a function value or passing arguments. Therefore, you should write an AST routine as a subroutine and use common blocks to pass arguments between an AST routine and the program that queues it.

  In some cases, a system service that queues an AST allows you to specify an argument for the AST routine (for example, SYS$GETJPI). If you choose to pass the argument, the AST routine must be written to accept the argument.
- Terminal I/O—If you try to access the terminal with language I/O statements using SYS$INPUT or SYS$OUTPUT, you may receive a redundant I/O error. You must establish another channel to the terminal by explicitly opening the terminal (or by using the SMG$ routines).
- Shared routines—An AST routine might invoke a subprogram that is also invoked by another program unit in the program. To prevent conflicts, a program unit should use the SYS$SETAST system service to disable AST interrupts before calling a routine that might be invoked by an AST. Once the shared routine has executed, the program unit can use the same service to reenable AST interrupts.
- Invocation—You should never directly call an AST routine as a subroutine or a function.
- Iteration—You should never allow an AST routine to be delivered iteratively.

The system service used to queue the AST routine determines whether the AST is delivered after a specified event or time.

- Event—The following system routines allow you to specify an AST routine to be delivered when the system routine completes:
  - LIB$SPAWN—Signals when the subprocess has been created.
  - SYS$ENQ and SYS$ENQW—Signal when the resource lock is blocking a request from another process.
  - SYS$GETDVI and SYS$GETDVIW—Indicate that device information has been received.

  - SYS$GETJPI and SYS$GETJPIW—Indicate that process information has been received.
  - SYS$GETSYI and SYS$GETSYIW—Indicate that system information has been received.
  - SYS$QIO and SYS$QIOW—Signal when the requested I/O is completed.
  - SYS$UPDSEC—Signals when the section file has been updated.
- Event—The SYS$SETPRA system service allows you to specify an AST to be delivered when the system detects a power recovery.
- Time—The SYS$SETIMR system service allows you to specify a time for the AST to be delivered.
- Time—The SYS$DCLAST system service delivers a specified AST immediately. This makes it an ideal tool for debugging AST routines.

If a program queues an AST and then exits before the AST is delivered, the AST is deleted from the queue. If a process is hibernating when an AST is delivered, the AST executes and the process continues hibernating. If a process is suspended when an AST is delivered, the AST executes as soon as the process is resumed. If more than one AST is delivered, they are executed in the order in which they were delivered.

Generally AST routines are used with the SYS$QIO or SYS$QIOW system service for handling CTRL/C, CTRL/Y, and unsolicited input. See Section 7.5 for more information and examples.

4.3 Specifying a Time for Program Execution

You can synchronize timed program execution in the following ways:

- Using one process to invoke an image in a subprocess or detached process at specified times.
- Placing entries in the system timer queue.

4.3.1 Using Processes for Timing

Create a subprocess or detached process to execute the image at a specified time. Then, wake the subprocess or detached process when it is time for the image to be executed. If you expect the parent process to exit before the program in the subprocess or detached process finishes executing, create a detached process rather than a subprocess.

You can use either system services or RTL routines for obtaining and reading time. They are summarized in Table 4-3. With these routines, you can determine the system time, convert it to an external time, and pass a time back to the system. The system services use the VMS default date format. With the RTL routines, you can use the default format or specify your own date format. However, if you are just using the time and date for program synchronization, using the VMS default format is probably sufficient.

When using the RTL routines to change date/time formats, initialization routines are required. Refer to the *VMS Run-Time Library Routines Volume* for more information.

Once the time is specified, use the wake-up routine SYS$SCHDWK to invoke the subprocess or detached process for execution.

Table 4–3 Time Statistics System Services

Routine	Description
SYS$GETTIM	Obtains the current date and time in 64-bit binary format
SYS$NUMTIM LIB$SYS_ASCTIM	Converts system date and time to numeric integer values
SYS$BINTIM	Converts a date and time from ASCII to system format
LIB$ADD_TIMES	Adds two quadword times
LIB$CONVERT_DATE_STRING	Converts an input date/time string to a VMS internal time
LIB$CVT_FROM_INTERNAL_TIME	Converts internal time to external time
LIB$CVTF_FROM_INTERNAL_TIME	Converts internal time to external time (F-floating value)
LIB$CVT_TO_INTERNAL_TIME	Converts external time to internal time
LIB$CVTF_TO_INTERNAL_TIME	Converts external time to internal time (F-floating value)
LIB$CVT_VECTIM	Converts 7-word vector to internal time
LIB$DAY	Obtains offset to current day from base time, in number of days
LIB$DATE_TIME	Obtains the date and time in user-specified format
LIB$FORMAT_DATE_TIME	Formats a date and/or time for output
LIB$FREE_DATE_TIME_CONTEXT	Frees date/time context
LIB$GET_DATE_FORMAT	Returns the user's specified date/time input format
LIB$GET_MAXIMUM_DATE_ LENGTH	Returns the maximum possible length of an output date/time string
LIB$GET_USERS_LANGUAGE	Returns the user's selected langauge
LIB$INIT_DATE_TIME_CONTEXT	Initializes the date/time context with a user-specified format
LIB$SUB_TIMES	Subtracts two quadword times

4.3.1.1 Specified Time

To execute a program at a specified time, use LIB$SPAWN to create a process that executes a command procedure containing two commands—the DCL command WAIT and the command that invokes the desired program. Since you do not want the parent process to remain in hibernation until the process executes, execute the process concurrently.

You can also use SYS$CREPRC to execute a program at a specified time. However, since a process created by SYS$CREPRC hibernates rather than terminates after executing the desired program, LIB$SPAWN is preferred unless you need a detached process.

Example 4–1 executes a program at a specified delta time. The parent program prompts the user for a delta time, equates the delta time to the symbol EXECUTE_TIME, and then creates a subprocess to execute the command procedure LATER.COM. LATER.COM uses the symbol EXECUTE_TIME as the parameter for the WAIT command. (You might also allow the user to enter an absolute time and have your program change it to a delta time by subtracting the current time from the specified time. Section 3.3 discusses time manipulation.)

Example 4–1 Executing a Program Using Delta Time

```
! Delta time
CHARACTER*17 TIME
INTEGER LEN
! Mask for LIB$SPAWN
INTEGER*4 MASK

! Declare status and library routine
INTEGER STATUS, LIB$SPAWN

! Get delta time
STATUS = LIB$GET_INPUT (TIME,
2                        'Time (delta): ',
2                        LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Equate symbol to TIME
STATUS = LIB$SET_SYMBOL ('EXECUTE_TIME',
2                         TIME(1:LEN))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Set the mask and call LIB$SPAWN
MASK = IBSET (MASK,0)            ! Execute subprocess concurrently
STATUS = LIB$SPAWN('@LATER',
2                  'DATA84.IN',
2                  'DATA84.RPT',
2                  MASK)

END
```

LATER.COM

```
$ WAIT 'EXECUTE_TIME'
$ RUN SYS$DRIVE0:[USER.MATH]CALC
$ DELETE/SYMBOL EXECUTE_TIME
```

4.3.1.2 Timed Intervals

To execute a program at timed intervals, you can use either LIB$SPAWN or SYS$CREPRC.

Using LIB$SPAWN

Using LIB$SPAWN, you can create a subprocess that executes a command procedure containing three commands: the DCL command WAIT, the command that invokes the desired program, and a GOTO command that directs control back to the WAIT command. Since you do not want the parent process to remain in hibernation until the subprocess executes, execute the subprocess concurrently. See Section 4.3.1.1 for an example of LIB$SPAWN.

Using SYS$CREPRC

Using SYS$CREPRC, create a detached process to execute a program at timed intervals as follows:

1 Create and hibernate a process—Use SYS$CREPRC to create a process that executes the desired program. Set the PRC$V_HIBER bit of the **stsflg** argument of the SYS$CREPRC system service to indicate that the created process should hibernate before executing the program.

2 Schedule a wakeup call for the created subprocess—Use the SYS$SCHDWK system service to specify the time at which the system should wake up the subprocess and a time interval at which the system should repeat the wakeup call.

Example 4–2 executes a program at timed intervals. The program creates a subprocess that immediately hibernates. (The identification number of the created subprocess is returned to the parent process so that it can be passed to SYS$SCHDWK.) The system wakes up the subprocess at 6:00 a.m. the morning of the 23rd (month and year default to system month and year) and every 10 minutes thereafter.

Example 4–2 Executing a Program at Timed Intervals

```
! SYS$CREPRC options and values
INTEGER OPTIONS
EXTERNAL PRC$V_HIBER
! ID of created subprocess
INTEGER CR_ID
! Binary times
INTEGER TIME(2),
2       INTERVAL(2)
    .
    .
    .
! Set the PRC$V_HIBER bit in the OPTIONS mask and
! create the process
OPTIONS = IBSET (OPTIONS, %LOC(PRC$V_HIBER))
STATUS = SYS$CREPRC (CR_ID,          ! PID of created process
2                    'CHECK',        ! Image
2                    ,,,,,
2                    'SLEEP',        ! Process name
2                    %VAL(4),        ! Priority
2                    ,,
2                    %VAL(OPTIONS)) ! Hibernate
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Translate 6:00 a.m. (absolute time) to binary
STATUS = SYS$BINTIM ('23-- 06:00:00.00', ! 6:00 a.m.
2                    TIME)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Translate 10 minutes (delta time) to binary
STATUS = SYS$BINTIM ('0 :10:00.00',      ! 10 minutes
2                    INTERVAL)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Schedule wakeup calls
STATUS = SYS$SCHDWK (CR_ID,          ! ID of created process
2                    ,
2                    TIME,           ! Initial wakeup time
2                    INTERVAL)       ! Repeat wakeup time
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
    .
    .
    .
```

4.3.2 Placing Entries in the System Timer Queue

When you use the system timer queue, you use the timer expiration to signal when an image is to be executed. You can use an event flag or AST for the actual signal. With this technique, you do not need a separate process to

control program execution. However, you do use up your process's quotas for ASTs and timer queue requests.

Use the system service SYS$SETIMR to place a request in the system timer queue. The format of this service is as follows:

SYS$SETIMR ([efn],daytim,[astadr],[reqidt])

Specifying the Starting Time

Specify the absolute or delta time at which you want the program to begin execution using the **daytim** argument. Use SYS$BINTIM to convert an ASCII time to the binary system format required for this argument.

Signaling Timer Expiration

Once the system has reached this time, the timer expires. To signal timer expiration, set an event flag in the **efn** argument or specify an AST routine to be executed in the **astadr** argument. Refer to Sections 4.1 and 4.2 for more information on using event flags and ASTs.

How Timer Requests Are Identified

The **reqidt** argument identifies each system time request uniquely. Then, if you need to cancel a request, you can refer to each request separately.

To cancel a timer request, use SYS$CANTIM.

4.4 Synchronous and Asynchronous System Services

A number of system services can be executed either synchronously or asynchronously (for example, SYS$GETJPI and SYS$GETJPIW). The "W" at the end of the system service name indicates the synchronous version of the system service.

The asynchronous version of a system service queues a request and returns control to your program. You can perform operations while the system service executes; however, do not attempt to access information returned by the service until the system service has completed.

Typically, you pass an event flag and an I/O status block to an asynchronous system service. When the system service completes, it sets the event flag and places the final status of the request in the I/O status block. Use the SYS$SYNCH system service to ensure that the system service has completed. You pass SYS$SYNCH the event flag and I/O status block that you passed to the asynchronous system service; SYS$SYNCH waits for the event flag to be set and then checks that the system service rather than some other program set the event flag by examining the I/O status block. If the I/O status block is still 0, SYS$SYNCH waits until the I/O status block is filled.

```
! Data structure for SYS$GETJPI
    .
    .
    .
INTEGER*4 STATUS,
2         FLAG,
2         PID_VALUE
! I/O status block
STRUCTURE /STATUS_BLOCK/
 INTEGER*2 JPISTATUS,
2          LEN
 INTEGER*4 ZERO /0/
END STRUCTURE
RECORD /STATUS_BLOCK/ IOSTATUS
    .
    .
    .
! Call SYS$GETJPI and wait for information
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$GETJPI (%VAL(FLAG),
2                    PID_VALUE,
2                    ,
2                    NAME_BUF_LEN,
2                    IOSTATUS,
2                    ,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
    .
    .
    .
STATUS = SYS$SYNCH (%VAL(FLAG),
2                   IOSTATUS)
IF (.NOT. IOSTATUS.JPISTATUS) THEN
  CALL LIB$SIGNAL (%VAL(IOSTATUS.JPISTATUS))
END IF

END
```

The synchronous version of a system service acts exactly as if you had used the asynchronous version followed immediately by a call to SYS$SYNCH. Regardless of whether you use the synchronous or asynchronous version of a system service, if you omit the **efn** argument, the service uses event flag 0.

4.5 Using the Lock Manager

The lock manager can be used by cooperating processes to synchronize access to a shared resource (for example, a file, program, or device). The lock manager itself does not ensure proper access to the resource; rather, the programs must respect the rules for using the lock manager.

The rules required for proper resource synchronization are as follows:

- The resource must always be referred to by an agreed-upon name.
- Access to the resource is always done by queueing a lock request with SYS$ENQ or SYS$ENQW.
- All lock requests that are placed in a wait queue must wait for access to the resource.

When the lock manager is used by a process and its subprocess, the program that created the subprocess should not exit until the subprocess has exited. To ensure that the parent does not exit before the subprocess, specify an event flag to be set when the subprocess exits (the **completion-efn** argument of LIB$SPAWN). Before exiting from the parent program, use SYS$WAITFR to ensure that the event flag is set. (You can suppress the logout message from the subprocess by using the SYS$DELPRC system service to delete the subprocess instead of allowing the subprocess to exit.)

The lock manager services are summarized in Table 4–4.

Table 4–4 Lock Manager Routines

Routines	Description
SYS$ENQ SYS$ENQW	Queue a new lock or lock conversion on a resource.
SYS$DEQ	Release locks and cancel lock requests.
SYS$GETLKI SYS$GETLKIW	Get information about the lock database.

4.5.1 Requesting a Lock

To request access to a resource, use the SYS$ENQ or SYS$ENQW system services to queue a lock request. (SYS$ENQ queues a lock request and returns; SYS$ENQW queues a lock request, waits until the lock is granted, and then returns.) The following lock modes allow a process to indicate the extent to which it is willing to share the resource:

- Null—No lock is requested; rather, it serves as a placeholder and indicator of future interest in the resource.
- Concurrent read—Read access to the requestor while maintaining write access to others.
- Concurrent write—Write access to the requestor while maintaining write access to others.
- Protected read—Read access to the requestor while allowing only read access to others.
- Protected write—Write access to the requestor while allowing only read access to others.
- Exclusive—Write access to the requestor and no access to others.

The format for SYS$ENQ is as follows:

```
SYS$ENQ ([efn], lkmode, lksb, [flags], [resnam], [parid], [astadr],
[astprm], [blkast], [acmode], nullarg)
```

For more complete information on the use of SYS$ENQ, refer to the *VMS System Services Reference Manual.*

4.5.2 Requesting a Null Lock

The program segment in Example 4–3 requests a null lock for the resource named TERMINAL. After the lock is granted, the program requests that the lock be converted to an exclusive lock. Note that, after SYS$ENQW returns, the program checks the status of the system service and the status returned in the lock status block to ensure that the request completed successfully. (The lock mode symbols are defined in the $LCKDEF module of the system macro library.)

Example 4–3 Requesting a Null Lock

```
! Define lock modes
INCLUDE '($LCKDEF)'
! Define lock status block
STRUCTURE /STATUS_BLOCK/
 INTEGER*2 LOCK_STATUS,
2          NULL
 INTEGER*4 LOCK_ID
END STRUCTURE
RECORD /STATUS_BLOCK/ IOSTATUS
    .
    .
    .
! Request a null lock
STATUS = SYS$ENQW (,
2                  %VAL(LCK$K_NLMODE),
2                  IOSTATUS,
2                  ,
2                  'TERMINAL',
2                  ,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. IOSTATUS.LOCK_STATUS)
2    CALL LIB$SIGNAL (%VAL(IOSTATUS.LOCK_STATUS))
! Convert the lock to an exclusive lock
STATUS = SYS$ENQW (,
2                  %VAL(LCK$K_EXMODE),
2                  IOSTATUS,
2                  %VAL(LCK$M_CONVERT),
2                  'TERMINAL',
2                  ,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. IOSTATUS.LOCK_STATUS)
2    CALL LIB$SIGNAL (%VAL(IOSTATUS.LOCK_STATUS))
```

4.6 Using the Parallel Processing Run-time Library Routines

The parallel processing (PPL$) facility consists of routines for synchronizing program processing in a master/slave hardware configuration. The routines provide for the following capabilities:

- Creating subprocesses
- Synchronizing program execution using spin locks
- Synchronizing program execution using semaphores

- Synchronizing program execution using barriers
- Setting up global sections of memory for shared use

To use the PPL$ routines, you must call the PPL$ initialization routine (PPL$INITIALIZE) that sets up data structures and memory areas required for PPL. Then, when use of the PPL$ routines is no longer required, you must free those data structures and memory areas with PPL$TERMINATE before exiting from the program.

Refer to the *VMS RTL Parallel Processing (PPL$) Manual* for more information.

4.6.1 Using Subprocesses

Once you have initialized the PPL environment, you can create one or more subprocesses to execute images. You may execute the same or different images within each subprocess. Even though you can create a subprocess with PPL$CREATE_PROCESS that will run outside of a PPL environment, you should limit its use to subprocesses within a PPL environment.

To delete one or more subprocesses created with PPL$CREATE_PROCESS, use PPL$DELETE_PROCESS.

4.6.2 Using Spin Locks

To ensure that only one process at a time can access a critical region or physical resource of a parallel task, you can use spin locks. A spin lock is a lock on a critical region where the lock constantly tests to determine whether access to the critical region is available. Because of the constant testing, this technique is CPU intensive. An alternative technique to ensure single access is to use semaphores. Refer to Section 4.6.3 for more information on using semaphores.

There are three spin lock routines, which are used as follows:

- PPL$CREATE_SPIN_LOCK—Creates and initializes a simple lock. An identifier is returned for subsequent reference to this spin lock.
- PPL$SEIZE_SPIN_LOCK—Acquires a spin lock that has been created with PPL$CREATE_SPIN_LOCK. Use the identifier returned by PPL$CREATE_SPIN_LOCK to refer to the lock you want to acquire.
- PPL$RELEASE_SPIN_LOCK—Releases a spin lock. Use the identifier returned by PPL$CREATE_SPIN_LOCK to refer to the lock you want to release. Once this routine has freed the lock, another process can acquire the lock.

4.6.3 Using Semaphores

Semaphores also synchronize access to a critical region or physical device by controlling the number of processes that have access. Unlike spinlocks, using semaphores is not CPU intensive.

There are two type of semaphores: a binary semaphore and a counting semaphore. A binary semaphore has a value of 0 and 1 and allows only one process to access a resource. A process can access the resource when the semaphore value is 1. A process waits for the resource when the semaphore is zero. A counting semaphore can have any positive value, thereby allowing you to control access to multiple resources.

The semaphore routines are as follows:

- PPL$CREATE_SEMAPHORE—Creates and initializes a semaphore and creates a waiting queue that keeps track of processes waiting for the semaphore.
- PPL$DECREMENT_SEMAPHORE—Decrements the value of a semaphore. If the value of the semaphore is zero, the process requesting the semaphore can be placed in a wait state until the semaphore value increases.
- PPL$INCREMENT_SEMAPHORE—Increments the value of a semaphore to indicate that the resource can be accessed. If there is a process waiting for the semaphore, PPL$INCREMENT_SEMAPHORE wakes up the process and removes it from the queue.
- PPL$RETURN_SEMAPHORE_VALUES—Returns the value of the requested semaphore.

4.6.4 Using Barrier Synchronization

Barrier synchronization specifies a point in a program that all parallel paths must reach before any are allowed to continue. Only one barrier can be set up within a program.

The barrier routines include the following:

- PPL$CREATE_BARRIER—Specifies the point that all paths must reach before continuation.
- PPL$WAIT_AT_BARRIER—Suspends execution of the program path until all program paths have reached the specified barrier.

Once you specify a barrier point, all program paths must call PPL$WAIT_AT_BARRIER in order to be included in the barrier synchronization.

4.7 Writing Applications for a VMS Multiprocessing Environment

Most application programs that run on a VMS uniprocessing system run without modification on a VMS multiprocessing system. However, those applications that access writable global sections or rely on process priority as a means of synchronizing tasks should be reexamined and modified according to the information contained in this section.

In addition, some applications may execute more efficiently on a multiprocessor if they are specifically adapted to a multiprocessing environment. Programmers may want to decompose an application into several processes and coordinate their activities by means of event flags or a shared region in memory. See the *VMS RTL Parallel Processing (PPL$) Manual* and the *Guide to Parallel Programming on VMS* for more information about performing these tasks.

System programmers, including those writing device drivers and user-written system services, should refer to other sections of this chapter for critical information about system synchronization techniques.

4.7.1 Writable Global Sections

A writable global section is an area of memory that can be accessed (read and modified) by more than one process. In a uniprocessor system, access to a global section by more than one process is automatically synchronized as follows:

- Only the currently executing process can access the global section.
- Only one process can be the currently executing process.

However, in the multiprocessing system, two or more processes can execute concurrently, one on each processor. As a result, it is possible that concurrently executing processes can simultaneously access the same locations in a writable global section. If such access occurs, information may be lost.

When writing an application program for a VMS multiprocessing system, you must use one of the following methods to ensure synchronized access to the global sections by multiple processes:

- Use interlocked instructions instead of ordinary instructions to control access to the writable global section. The seven interlocked VAX MACRO instructions are as follows:
  - BBCCI—Branch on Bit Clear and Clear, Interlocked
  - BBSSI—Branch on Bit Set and Set, Interlocked
  - ADAWI—Add Aligned Word, Interlocked
  - INSQTI—Insert into Queue Tail, Interlocked
  - INSQHI—Insert into Queue Head, Interlocked
  - REMQTI—Remove from Queue Tail, Interlocked
  - REMQHI—Remove from Queue Head, Interlocked
- Use VMS system services to control access to the writable global section.

Check existing programs that use writable global sections to ensure that proper synchronization techniques are in place. Review the program code itself; do not rely on testing alone because an instance of simultaneous access by more than one process to a location in a writable global section is rare.

If an application must use queue instructions to control access to writable global sections, ensure that it uses interlocked queue instructions.

4.7.2 Synchronization Using Process Priority

In some applications (usually real-time applications), a number of processes are used to perform a series of tasks. In such applications, the sequence in which a process executes can be controlled or synchronized by means of process priority. The basic method of synchronization by priority involves executing the process with the highest priority while preventing all other processes from executing.

Because each processor in a VMS multiprocessing system, when idle, schedules its own workload, it is impossible to prevent all other processes in the system from executing. Moreover, because the scheduler guarantees only that the highest priority process be scheduled at any given time, it is not a certainty that another processor in the system is executing the next highest priority process.

Thus, application programs that use the method of synchronization by process priority must be modified to use a different serialization method before they will run correctly in a VMS multiprocessing system.

4.8 Passing Control to Another Image

The RTL routines LIB$DO_COMMAND and LIB$RUN_PROGRAM allow you to invoke the next image from the current image. That is, they allow you to perform image run-down for the current image and pass control to the next image without returning to DCL command level. Which routine you use depends on whether the next image is a command image or a noncommand image.

4.8.1 Invoking a Command Image

A command image is invoked at DCL command level with the appropriate DCL command. The following command executes the command image associated with the DCL command COPY:

```
$ COPY DATA.TMP APRIL.DAT
```

To pass control from the current image to a command image, use the RTL routine LIB$DO_COMMAND. If LIB$DO_COMMAND executes successfully, control is not returned to the invoking image and statements following the LIB$DO_COMMAND statement are not executed. The following statement causes the current image to exit and executes the DCL command just shown:

```
      .
      .
      .
STATUS = LIB$DO_COMMAND ('COPY DATA.TMP APRIL.DAT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

To execute a number of DCL commands, specify a DCL command procedure. The following statement causes the current image to exit and executes the DCL command procedure [STATS.TEMP]CLEANUP.COM:

```
      .
      .
      .
STATUS = LIB$DO_COMMAND ('@[STATS.TEMP]CLEANUP')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

4.8.2 Invoking a Noncommand Image

A noncommand image is invoked at DCL command level with the DCL command RUN. The following command executes the noncommand image [STATISTICS.TEMP]TEST.EXE:

```
$ RUN [STATISTICS.TEMP]TEST
```

To pass control from the current image to a noncommand image, use the run-time library routine LIB$RUN_PROGRAM. If LIB$RUN_PROGRAM executes successfully, control is not returned to the invoking image and statements following the LIB$RUN_PROGRAM statement are not executed. The following program segment causes the current image to exit and passes control to the noncommand image [STATISTICS.TEMP]TEST.EXE on the default disk:

```
      .
      .
      .
STATUS = LIB$RUN_PROGRAM ('[STATISTICS.TEMP]TEST.EXE')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

5 Shareable Resources

The VMS operating system provides the following techniques for sharing data and program code among programs:

- DCL symbols and logical names
- Libraries
- Shareable images
- Global sections
- Common blocks installed in a shareable image
- VMS RMS shared files

Symbols and logical names are also used for intraprocess and interprocess communication; therefore, they are discussed in Chapter 3.

Libraries and shareable images are used for sharing program code.

Global sections, common blocks stored in shareable images, and VMS RMS shared files are used for sharing data. Using common blocks for inter-process communication is also possible. Refer to Chapter 3.

5.1 Sharing Program Code

To share code among programs, you can use the following VMS resources:

- Text, macro, or object libraries which store sections of code. Text and macro libraries store source code; object libraries store object code. You can create and manage libraries using LIBRARIAN. Refer to the *VMS Librarian Utility Manual* for complete information on using LIBRARIAN.
- Shareable images, which are images that have been compiled and linked, but cannot be run independently. These images can also be stored in libraries.

5.1.1 Object Libraries

Object libraries can be used to store frequently used routines, thereby avoiding repeated recompiling, minimizing the number of files you must maintain, and simplifying the linking process. The source code for the object modules can be in any VAX-supported language and the object modules can be linked with any other modules written in any VAX-supported language.

Use the OLB file extension for any object library. All modules stored in an object library must have the file extension OBJ.

5.1.1.1 System- and User-Defined Default Object Libraries

The VMS operating system provides a default system object library, STARLET.OLB. You can also define one or more default object libraries to be automatically searched before the system object library. The logical names for the default object libraries are LNK$LIBRARY and LNK$LIBRARY_1 through LNK$LIBRARY_999. To use one of these default libraries, define the logical name first. They are searched sequentially starting at LNK$LIBRARY. Do not skip any numbers. If you store object modules in the default libraries, you do not have to specify them at link time. However, you do have to maintain and manage them as you would any library.

The following example defines the library in the file PROCEDURES.OLB (the file type defaults to OLB) in $DISK1:[DEV] as a default user library:

```
$ DEFINE LNK$LIBRARY $DISK1:[DEV]PROCEDURES
```

5.1.1.2 How the Linker Searches Libraries

When the linker is resolving global symbol references, it searches user default libraries at the process level first, then libraries at the group and system level. Within levels, the library defined as LNK$LIBRARY is searched first, then LNK$LIBRARY_1, LNK$LIBRARY_2, and so on.

5.1.1.3 Creating an Object Library

To create an object library, invoke the Librarian Utility by entering the LIBRARY command with the /CREATE qualifier and the name you are assigning the library. The following example creates a library in a file named INCOME.OLB (OLB, the default file type, means object library):

```
$ LIBRARY/CREATE INCOME
```

5.1.1.4 Managing an Object Library

To add or replace modules in a library, enter the LIBRARY command with the /REPLACE qualifier followed by the name of the library (first parameter) and the names of the files containing the module or modules (second parameter). After you put an object module or modules in a library, you can delete the object file. The following example adds or replaces the modules from the object file named GETSTATS.OBJ to the object library named INCOME.OLB and then deletes the object file:

```
$ LIBRARY/REPLACE INCOME GETSTATS
$ DELETE GETSTATS.OBJ;*
```

You can examine the contents of an object library with the /LIST qualifier. Use the /ONLY qualifier to limit the display. The following command displays all the modules in INCOME.OLB that start with GET:

```
$ LIBRARY/LIST/ONLY=GET* INCOME
```

Use the /DELETE qualifier to delete a library module and the /EXTRACT qualifier to re-create an object file. If you delete many modules, you should also compress (/COMPRESS) and purge (PURGE command) the library. Note that the /ONLY, /DELETE, and /EXTRACT qualifiers require the names of modules—not file names—and that the names are specified as qualifier values, not parameter values.

5.1.2 Text and Macro Libraries

Any frequently used routine can be stored in libraries as source code. Then, when you need the routine, it can be called in from your source program.

Source code modules are stored in text libraries. The file extension for a text library is TLB.

When using VAX MACRO assembly language, any source code module can be stored in a macro library. The file extension for a macro library is MLB. Any source code module stored in a macro library must have the file extension MAR.

You also use LIBRARIAN to create and manage text and macro libraries. Refer to Sections 5.1.1.3 and 5.1.1.4 for a summary of LIBRARIAN commands.

5.2 Shareable Images

A shareable image is a nonexecutable image that can be linked with executable images. If you have a program unit that is invoked by more than one program, linking it as a shareable image provides the following benefits:

- Saves disk space—The executable images to which the shareable image is linked do not physically include the shareable image. Only one copy of the shareable image exists.
- Simplifies maintenance—If you use transfer vectors and the GSMATCH option (see Section 5.2.4), you can modify, recompile, and relink a shareable image without having to relink any executable image that is linked with it.

Shareable images can also save memory provided that they are installed as shared images (see Section 5.2.7).

5.2.1 Transfer Vectors

A transfer vector is placed at the beginning of a shareable image to point to a program unit in that shareable image. Typically, a shareable image contains one program unit and one transfer vector. If you have more than one program unit in a shareable image, include a transfer vector for each program unit. The following example shows a macro program unit that contains two transfer vectors, one for GET_1_STAT and one for GET_STATS:

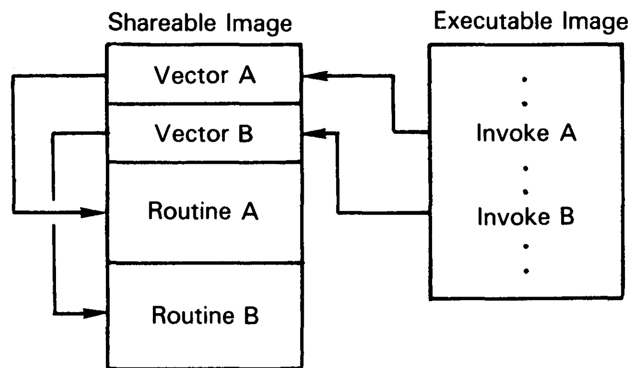
XGETSTATS.MAR

```
.TITLE X_GET_STATS

.TRANSFER       GET_1_STAT
.MASK   GET_1_STAT
JMP     L^GET_1_STAT+2

.TRANSFER       GET_STATS
.MASK   GET_STATS
JMP     L^GET_STATS+2

.END
```

5.2.1.1 Why Use Transfer Vectors?

You should always use transfer vectors; they allow you to modify a shareable image without relinking any executable image that references the shareable image. When you link a shareable image to produce an executable image, the linker resolves a reference to a program unit in that shareable image by using the address of the transfer vector for that program unit as shown in Figure 5–1. If you modify a program unit in a shareable image, the starting address of one or more program units may change; relinking the shareable image updates each transfer vector to point to the correct starting address of its associated program unit. Since the addresses of the transfer vectors have not been modified, executable images linked with the shareable image do not have to be relinked.

Figure 5–1 How the Linker Uses Transfer Vector Address



5.2.1.2 Deleting Transfer Vectors

You should not delete a transfer vector from a shareable image that contains more than one transfer vector. Deleting one transfer vector may change the addresses of other transfer vectors in the shareable image. If you change the address of a transfer vector, you have to relink each executable image that references that shareable image. If you must delete a program unit from a shareable image containing more than one program unit, create a dummy program unit with the same name, such as the one for GET_1_STAT in the following example:

GET1STAT.FOR

```
FUNCTION GET_1_STAT (ROW,
2                    COLUMN,
2                    STAT)
! Dummy routine
END
```

Compile the dummy program unit and relink the shareable image. In the new version of the shareable image, the transfer vector for the "deleted" program unit points to the dummy program unit.

5.2.2 GSMATCH Option

The GSMATCH option allows you to specify whether an executable image linked with a shareable image can access a modified shareable image. The GSMATCH option must be specified in an options file (use the /OPTIONS qualifier of the LINK command; for details, see the description of the linker in the *VMS Linker Utility Manual.*

When an executable image attempts to access a shareable image at run time, the system examines the GSMATCH option specified by the shareable image that was originally linked with the executable image. The following keywords may be specified with the GSMATCH option:

- LEQUAL—If the minor ID of the original shareable image is less than or equal to the minor ID of the shareable image that the executable image is attempting to access, the system allows the executable image to access the shareable image.
- EQUAL—If the minor ID of the original shareable image is equal to the minor ID of the shareable image that the executable image is attempting to access, the system allows the executable image to access the shareable image. (Default if no GSMATCH option is specified.)
- ALWAYS—The system allows the executable image to access the shareable image regardless of the major ID or minor ID.

To examine the major and minor ID values of a shareable image, use the command LINK/MAP/FULL to produce a listing of the image that includes the GSMATCH option.

5.2.3 UNIVERSAL Option

A universal symbol is a global symbol in a shareable image that can be referenced outside the shareable image. A transfer vector, in addition to creating a pointer to a program unit, makes the name of that program unit a universal symbol. To make a symbol other than a program unit name a universal symbol, use the UNIVERSAL option in an options file (use the /OPTIONS qualifier of the LINK command; for details, see the description of the linker in the *VMS Linker Utility Manual.*)

A reference to a universal symbol is resolved at link time as an offset from the beginning of the defining routine. This implies that, if you modify the routine that defines a universal symbol, you must relink that routine to correct the offset to the universal symbol. Since universal symbols created by transfer vectors are always at the beginning of the defining module (see Section 5.2.1), relinking is necessary only if the universal symbol is created using the UNIVERSAL option.

5.2.4 Creating Shareable Images

To create a shareable image, follow these steps:

1. Compile the object modules—Write and compile the program unit to be shared by your different programs. In general, you want to produce a shareable image that executes one program unit. If that program unit invokes any other subprograms, they also must be included in the shareable image.

2. Write the transfer vector—Write a transfer vector for each program unit in the shareable image (Section 5.2.1 discusses transfer vectors); transfer vectors must be written in VAX MACRO. The following template contains one transfer vector; repeat the three middle statements for each additional transfer vector:

   ```
   .TITLE  vector-name
   .TRANSFER        routine-name
   .MASK    routine-name
   JMP      L^routine-name+2
   .END
   ```

   Naming the transfer vector file with a name similar to that of the program unit's source file makes the transfer vector file easier to find. The transfer vector file and the associated language program units should not have identical names because, after compiling the source files, you'll have two or more object modules with the same name. The following VAX MACRO transfer vector file is for the program unit GET_1_STAT:

   XGET1STAT.MAR

   ```
   .TITLE X_GET_1_STAT
   .TRANSFER       GET_1_STAT
   .MASK   GET_1_STAT
   JMP     L^GET_1_STAT+2
   .END
   ```

   Compiling the transfer vector with the MACRO command produces an object module named X_GET_1_STAT in the file XGET1STAT.OBJ as follows:

   ```
   $ MACRO XGET1STAT
   ```

3. Write an options file for the linker—Use the CLUSTER option to place the transfer vector at the beginning of the shareable image. Use the GSMATCH option to specify whether an executable image linked with the shareable image can access a modified version of that shareable image without relinking.

   The CLUSTER option takes the following form:

   CLUSTER=cluster-name,,,filename

   The cluster name is up to you. The file name is that of the object file containing the transfer vector.

   The GSMATCH option takes the following form:

   GSMATCH=keyword,major_id,minor_id

Typically, when you create a shareable image, you use the LEQUAL keyword, specifying any integer values for the major and minor IDs; when you update that shareable image, you use the LEQUAL keyword, specifying the same major ID and incrementing the minor ID by 1. This use of LEQUAL allows an executable image to access a newer version of the shareable image without relinking, but prevents the executable image from accessing an older version of the shareable image (see Section 5.2.2 for more information).

When you create the shareable image GET1STAT, you could specify the following options files:

GET1STAT.OPT

```
CLUSTER=X_GET_1_STAT,,,XGET1STAT
GSMATCH=LEQUAL,1,100
```

When you update that shareable image, you would change the GSMATCH option as follows:

GET1STAT.OPT

```
CLUSTER=X_GET_1_STAT,,,XGET1STAT
GSMATCH=LEQUAL,1,101
```

4. Link to produce the shareable image—Use the /SHAREABLE qualifier of the LINK command to create a shareable image specifying the object modules and the options file as input to the linker. The following command produces a shareable image named GET1STAT.EXE from the object module GET1STAT.OBJ and the options file GET1STAT.OPT:

```
$ LINK/SHAREABLE GET1STAT,GET1STAT/OPTION
```

GET1STAT.OPT

```
CLUSTER=X_GET_1_STAT,,,XGET1STAT
GSMATCH=LEQUAL,1,100
```

Once you have created the shareable image, you can delete the object modules GET1STAT.OBJ and XGET1STAT.OBJ.

When you link a shareable image, references to global symbols must be resolved by including the module that defines the symbol in the link operation. For example, to create a shareable image from the program unit GET_STATS, which references the program unit GET_1_STAT, you must specify both GETSTATS.OBJ (the file containing GET_STATS) and GET1STAT.OBJ (the file containing GET_1_STAT) as input to the linker. The following command creates the shareable image GETSTATS.EXE. (XGETSTATS contains transfer vectors for both GET_STATS and GET_1_STAT.)

```
$ LINK/SHAREABLE GETSTATS,GET1STAT,GETSTATS/OPTION
```

GETSTATS.OPT

```
CLUSTER=X_GET_STATS,,,XGETSTATS
GSMATCH=LEQUAL,1,100
```

5.2.5 Shareable Image Libraries

In any large development effort, you should keep the program units in libraries (either object module or shareable image) to simplify maintenance and the linking process. Shareable image libraries, also called *shareable image symbol table libraries*, are different from object libraries in that the symbol table of the shareable image, not the shareable image itself, is placed into the shareable image library; therefore, do **not** delete a shareable image after placing it in a shareable image library.

The following commands create the shareable images GET1STAT.EXE and GETSTATS.EXE, placing them into the shareable image library INCOMESHR. OLB is the default file type for both shareable image and object module libraries. INCOMESHR is named in the second link command to resolve the reference to GET_1_STAT in GET_STATS.

```
$ LINK/SHAREABLE GET1STAT,GET1STAT/OPTION
$ LIBRARY/SHAREABLE/REPLACE INCOMESHR GET1STAT
$ LINK/SHAREABLE GETSTATS,GETSTATS/OPTION,INCOMESHR/LIBRARY
$ LIBRARY/SHAREABLE/REPLACE INCOMESHR GETSTATS
```

If you attempt to create GETSTATS.EXE before GET1STAT.EXE, the linker cannot resolve the reference to GET_1_STAT and displays the following warning message:

```
%LINK-W-USEUNDEF, 1 undefined symbol:
%LINK-I-UDFSYM,        GET_1_STAT
```

5.2.5.1 Adding or Replacing Shareable Images

To add or replace a shareable image in a shareable image library, enter the LIBRARY command with the /SHAREABLE and /REPLACE qualifiers followed by the name of the library (first parameter) and the file name of the shareable image (second parameter). The file type of the shareable image defaults to EXE. The following command enters the symbol table of the shareable image GET1STAT.EXE into the shareable library INCOMESHR.OLB:

```
$ LIBRARY/SHAREABLE/REPLACE INCOMESHR GET1STAT
```

5.2.5.2 Listing or Deleting Shareable Images

You can examine shareable image libraries with the /LIST qualifier of the LIBRARY command. You can delete shareable images from a shareable image library with the /DELETE qualifier of the LIBRARY command.

5.2.6 Linking Shareable Images

To specify a shareable image as input to the linker, you must specify either of the following: (1) the name of the shareable image library containing the symbol table of the shareable image (use the /LIBRARY qualifier to identify a library file) or (2) an options file that contains the name of the shareable image file (use the /SHAREABLE qualifier in the options file to identify a shareable image file). A shareable image file must be specified in an options file because a /SHAREABLE qualifier on the LINK command line is interpreted as a command qualifier that creates a shareable image.

The following command links the object module INCOME.OBJ with the library INCOME.OLB, and the shareable images GETSTATS.EXE and GET1STAT.EXE:

```
$ LINK INCOME,INCOME/OPTION,INCOME/LIBRARY
```

INCOME.OPT

```
GETSTATS/SHAREABLE
GET1STAT/SHAREABLE
```

The following command links the object module INCOME.OBJ, the object module library INCOME.OLB, and the shareable image library INCOMESHR.OLB to produce an executable image in the file INCOME.EXE:

```
$ LINK INCOME,INCOME/LIBRARY,INCOMESHR/LIBRARY
```

5.2.6.1 Default File Type and Location of Shareable Images

At link time, a shareable image is assumed to be in SYS$SHARE and to have a file type of EXE. Therefore, if you have not copied the shareable image over to SYS$SHARE, you must define a logical name that equates the name of the shareable image file to its full file specification.

The executable image INCOME.EXE created in the previous example references the shareable image files GETSTATS.EXE and GET1STAT.EXE. If these shareable images are in SYS$SHARE, you can execute INCOME as shown below:

```
$ RUN INCOME
```

5.2.6.2 Alternate Location of Shareable Images

However, if these shareable image files are in another directory, you must create logical names that associate the file names with the full file specifications. For example, if GETSTATS.EXE and GET1STAT.EXE are in the directory [INCOME.DEVELOP] on the disk $DISK1, define logical names for the files before executing INCOME.

```
$ DEFINE GETSTATS $DISK1:[INCOME.DEVELOP]GETSTATS
$ DEFINE GET1STAT $DISK1:[INCOME.DEVELOP]GET1STAT
$ RUN INCOME
```

If you attempt to execute INCOME without defining the logical names, the following messages are displayed (by default, SYS$SHARE translates to SYS$SYSROOT:[SYSLIB]):

```
%DCL-W-ACTIMAGE, error activating image GETSTATS
-CLI-E-IMAGEFNF, image file not found
                 SYS$SYSROOT:[SYSLIB]GETSTATS.EXE
```

In general, while you are developing a program that uses shareable images, you should leave the shareable images in your development directory and define the logical names each time you begin work on the program. If you are working on the program over a number of sessions, you may want to put the necessary logical name definitions in your LOGIN.COM file. Once the shareable images are working, you can move them into SYS$SHARE and delete the logical name definitions from LOGIN.COM.

5.2.7 Shared Images

To allow executable images to share a single copy of the shareable image in memory, install the shareable image as a *shared image.*

5.2.7.1 Creating a Shared Image

To install a shareable image as shared, follow the steps described in Chapter 6 for installing an image as privileged, but, instead of specifying the /PRIVILEGED qualifier, specify the /SHARED qualifier.

Perform the following steps to install a program as a shared image:

1. Enter the DCL command SET PROCESS/PRIVILEGE=CMKRNL to give yourself CMKRNL privilege.

Note: **You must have CMKRNL privilege in order to use the Install Utility (INSTALL).**

2. Enter the INSTALL command at the $ prompt to invoke the interactive Install Utility.
3. When the *INSTALL>* prompt appears, enter the following command:

   ```
   CREATE file-specification /SHARED
   ```

   Specify the complete file specification of the file containing the executable program (file type defaults to EXE).
4. Press the RETURN key. The Install Utility installs your program as a shared image and reissues the *INSTALL>* prompt.
5. Enter the EXIT command to exit from the Install Utility.
6. Enter the DCL command SET PROCESS/PRIVILEGE=NOCMKRNL to remove the CMKRNL privilege.

5.2.7.2 If the Shared Image Is in Memory

When an executable image linked with a shared image accesses the shared image, the executable image uses that copy.

5.2.7.3 If the Shared Image Is Not in Memory

If a copy of the shared image is not in memory, the executable image copies the shared image into memory. Unless the shareable image is likely to be accessed by more than one executable image at a time, do not bother to install the shareable image as a shared image.

5.3 Symbols

Symbols are names that represent locations (addresses) in virtual memory. More precisely, a symbol's value is the address of the first, or low-order, byte of a defined area of virtual memory, while the characteristics of the defined area provide the number of bytes referred to. For example, if you define TOTAL_HOUSES as an integer, the symbol TOTAL_HOUSES is assigned the address of the low-order byte of a 4-byte area in virtual memory. Some system components (for example, the debugger) permit you to refer to areas of virtual memory by their actual addresses, but symbolic references are always recommended.

5.3.1 Defining Symbols

A symbolic name can consist of up to 31 letters, digits, underscores, and dollar signs. Uppercase and lowercase letters are equivalent. By convention, dollar signs are restricted to symbols used in system components. (If you do not use the dollar sign in your symbolic names, you will never accidentally duplicate a system-defined symbol.)

5.3.2 Local and Global Symbols

Symbols are either local or global in scope. A local symbol can only be referenced within the program unit in which it is defined. Local symbol names must be unique among all other local symbols within the program unit, but not within other program units in the program. References to local symbols are resolved at compile time.

A global symbol can be referenced outside the program unit in which it is defined. Global symbol names must be unique among all other global symbols within the program. References to global symbols are not resolved until link time.

References to global symbols in the executable portion of a program unit are usually invocations of subprograms. If you reference a global symbol in any other capacity (as an argument or data value—see the following paragraph), you must define the symbol as external or intrinsic in the definition portion of the program unit.

System facilities, such as the Message Utility and the VAX MACRO assembler, use global symbols to define data values.

The following program segment shows how to define and reference a global symbol, RMS$_EOF (a condition code that may be returned by LIB$GET_INPUT):

```
CHARACTER*255   NEW_TEXT
INTEGER         STATUS
INTEGER*2       NT_SIZ
INTEGER         LIB$GET_INPUT
EXTERNAL        RMS$_EOF
STATUS = LIB$GET_INPUT (NEW_TEXT,
2                       'New text: ',
2                       NT_SIZ)
IF ((.NOT. STATUS) .AND.
2   (STATUS .NE. %LOC (RMS$_EOF))) THEN
  CALL LIB$SIGNAL (RETURN_STATUS BY VALUE)
END IF
```

5.3.3 Resolving Global Symbols

References to global symbols are resolved by including the module that defines the symbol in the link operation. When the linker encounters a global symbol, it uses the following search alogorithm to find the defining module:

1. Explicitly named modules and libraries—Generally used to resolve user-defined global symbols, such as subprogram names and condition codes. These modules and libraries are searched in the order in which they are specified.

2 System default libraries—Generally used to resolve system-defined global symbols, such as procedure names and condition codes.

3 User default libraries—Generally used to avoid explicitly naming libraries, thereby simplifying linking.

If the linker cannot find the symbol, the symbol is said to be unresolved, and a warning results. You can run an image containing unresolved symbols. The image runs successfully as long as it does not access any unresolved symbol. For example, if your code calls a subroutine but the subroutine call is not executed, the image runs successfully.

If an image accesses an unresolved global symbol, results are unpredictable. Usually the image fails with an access violation (attempting to access a physical memory location outside those assigned to the program's virtual memory addresses).

5.3.3.1 Explicitly Named Modules and Libraries

You can resolve a global symbol reference by naming the defining object module in the link command. For example, if the program unit INCOME references the subprogram GET_STATS, you can resolve the global symbol reference when you link INCOME by including the file containing the object module for GET_STATS, as follows:

```
$ LINK INCOME, GETSTATS
```

If the modules that define the symbols are in an object library, name the library in the link operation. In the following example, the GET_STATS module resides in the object module library INCOME.OLB:

```
$ LINK INCOME,INCOME/LIBRARY
```

5.3.3.2 System Default Libraries

Link operations automatically check the system object and shareable image libraries for any references to global symbols not resolved by your explicitly named object modules and libraries. The system object and shareable image libraries include the entry points for the RTL routines and system services, condition codes, and other system-defined values. Invocations of these modules do not require any explicit action by you at link time.

5.3.3.3 User Default Libraries

If you write general-purpose procedures or define general-purpose symbols, you can place them in a user default library. (You can also make your development library a user default library.) In this way, you can link to the modules containing these procedures and symbols without explicitly naming the library in the DCL LINK command. To name a single user library, equate the file name of the library to the logical name LNK$LIBRARY. For subsequent default libraries, use the logical names LNK$LIBRARY_1 through LNK$LIBRARY_999, as described in Section 5.1.1.

5.3.3.4 Making a Library Available for System-wide Use

To make a library available to everyone using the system, define it at the system level. To restrict use of a library or to override a system library, define the library at the process or group level. The following command line defines the default user library at the system level:

```
$ DEFINE/SYSTEM LNK$LIBRARY $DISK1:[DEV]PROCEDURES
```

5.3.3.5 Macro Libraries

Some system symbols are not defined in the system object and shareable image libraries. In such cases, the *VMS System Services Volume* and *VMS Run-Time Library Routines Volume* note that the symbols are defined in the system macro library and tell you the name of the macro containing the symbols. To access these symbols, you must first assemble a macro routine with the following source code. The keyword GLOBAL must be in uppercase. The .TITLE directive is optional, but recommended.

```
.TITLE  macro-name
macro-name          GLOBAL
    .
    .
    .
.END
```

The following example is a macro program that includes two system macros:

LBRDEF.MAR

```
.TITLE $LBRDEF
$LBRDEF GLOBAL
$LHIDEF GLOBAL
.END
```

Assemble the routine containing the macros with the MACRO command. You can place the resultant object modules in a default library or in a library that you specify in the LINK command, or you can specify the object modules in the LINK command. The following example places the $LBRDEF and $LHIDEF modules in a library before performing a link operation:

```
$ MACRO LBRDEF
$ LIBRARY/REPLACE INCOME LBRDEF
$ DELETE LBRDEF.OBJ;*
$ LINK INCOME,INCOME/LIBRARY
```

The following LINK command uses the object file directly:

```
$ LINK INCOME,LBRDEF,INCOME/LIBRARY
```

5.3.4 Sharing Data

Typically, you use an installed common block for interprocess communication or for allowing two or more processes to access the same data simultaneously. However, you must have CMKRNL privilege to install the common block. If you do not have CMKRNL privilege, global sections allow you to perform the same operations.

5.3.4.1 Installed Common Blocks

To share data among processes using a common block, you must install the common block as a shared shareable image and link each program that references the common block against that shareable image.

To install a common block as a shared image:

1. Define a common block—Write a program that declares the variables in the common block and defines the common block. This program should not contain executable code. The following VAX FORTRAN program defines a common block:

INC_COMMON.FOR

```
INTEGER TOTAL_HOUSES
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048)
COMMON /INCOME_DATA/ TOTAL_HOUSES,
2                    PERSONS_HOUSE,
2                    ADULTS_HOUSE,
2                    INCOME_HOUSE

END
```

2 Create the shareable image—Compile the program containing the common block. Use the LINK/SHAREABLE command to create a shareable image containing the common block.

```
$ FORTRAN INC_COMMON
$ LINK/SHAREABLE INC_COMMON
```

3 Install the shareable image—Use the DCL command SET PROCESS /PRIVILEGE to give yourself CMKRNL privilege (required for use of the Install Utility). Use the DCL command INSTALL to invoke the interactive Install Utility. When the *INSTALL>* prompt appears, type CREATE, followed by the complete file specification of the shareable image that contains the common block (file type defaults to EXE) and the qualifiers /WRITEABLE and /SHARED. The Install Utility installs your shareable image and reissues the *INSTALL>* prompt. Type EXIT to exit. Remember to remove CMKRNL privilege. (For complete documentation of the Install Utility, see the *VMS Install Utility Manual*.)

The following example shows how to install a shareable image:

```
$ SET PROCESS/PRIVILEGE=CMKRNL
$ INSTALL
INSTALL> CREATE DISK$USER:[INCOME.DEV]INC_COMMON-
_INSTALL> /WRITEABLE/SHARED
INSTALL> EXIT
$ SET PROCESS/PRIVILEGE=NOCMKRNL
```

Note: A disk containing an installed image cannot be dismounted. To remove an installed image, invoke the Install Utility and type DELETE followed by the complete file specification of the image. The DELETE subcommand does not delete the file from the disk; it removes the file from the list of known installed images.

Perform the following steps to write or read the data in an installed common block from within any program:

1 Include the same variable and common block definitions in the program.

2 Compile the program.

3 Link the program against the shareable image that contains the common block. (Linking against a shareable image requires an options file.)

```
$ LINK INCOME, DATA/OPTION
$ LINK REPORT, DATA/OPTION
```

DATA.OPT

```
INC_COMMON/SHAREABLE
```

4 Execute the program.

In the previous series of examples, the two programs INCOME and REPORT access the same area of memory through the installed common block INCOME_DATA (defined in INC_COMMON.FOR).

Typically, programs accessing shared data use common event flag clusters to synchronize read and write access to the data. Refer to Chapter 4 for more information on using event flags for program synchronization.

5.3.4.2 Global Sections

To share data using global sections, each process that plans to access the data includes a common block of the same name, which contains the variables for the data. The first process to reference the data declares the common block as a global section and, optionally, maps data to the section. (Data in global sections, as in private sections, must be page aligned; see Section 8.3 for instructions.)

To create a global section, invoke SYS$CRMPSC as described in Section 8.3, and add the following:

- Additional argument—Specify the name of the global section (argument 5). A program uses this name to access a global section.
- Additional flag—Set the SEC$V_GBL bit of the **flags** argument to indicate that the section is a global section.

As other programs need to reference the data, each can use either SYS$CRMPSC or SYS$MGBLSC to map data into the global section. If you know that the global section exists, best practice is to use the SYS$MGBLSC system service.

The format for SYS$MGBLSC is as follows:

SYS$MGBLSC (inadr,[retadr],[acmode],[flags],gsdnam,[ident],[relpag])

Refer to the *VMS System Services Reference Manual* for complete information on this system service.

In Example 5–1, one image, DEVICE.FOR, passes device names to another image, GETDEVINF.FOR. GETDEVINF.FOR returns the process name and the terminal associated with the process that allocated each device. The two processes use the global section GLOBAL_SEC to communicate. GLOBAL_SEC is mapped to the common block named DATA, which is page aligned by the options file DATA.OPT. Event flags are used to synchronize the exchange of information. UFO_CREATE.FOR, DATA.OPT, and DEVICE.FOR are included here for easy reference. Refer to Section 8.3 if you have questions about either of these programs.

Example 5–1 Interprocess Communication Using Global Sections

```
!UFO_CREATE.FOR
    .
    .
    .
INTEGER FUNCTION UFO_CREATE (FAB,
2                             RAB,
2                             LUN)

! Include RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'

! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN

! Declare channel
INTEGER*4 CHAN
COMMON /CHANNEL/ CHAN

! Declare status variable
INTEGER STATUS

! Declare system procedures
INTEGER SYS$CREATE

! Set useropen bit in the FAB options longword
FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_UFO
! Open file
STATUS = SYS$CREATE (FAB)

! Read channel from FAB status word
CHAN = FAB.FAB$L_STV

! Return status of open operation
UFO_CREATE = STATUS

END
```

DATA.OPT

```
PSECT_ATTR = DATA, PAGE
```

DEVICE.FOR

```
! Define global section flags
INCLUDE '($SECDEF)'
! Mask for section flags
INTEGER SEC_MASK
```

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Interprocess Communication Using Global Sections

```
! Logical unit number for section file
INTEGER INFO_LUN
! Channel number for section file
INTEGER SEC_CHAN
COMMON /CHANNEL/ SEC_CHAN
! Length for the section file
INTEGER SEC_LEN
! Data for the section file
CHARACTER*12 DEVICE,
2            PROCESS
CHARACTER*6 TERMINAL
COMMON /DATA/ DEVICE,
2             PROCESS,
2             TERMINAL
! Location of data
INTEGER PASS_ADDR (2),
2       RET_ADDR (2)
! Two common event flags
INTEGER REQUEST_FLAG,
2       INFO_FLAG
DATA REQUEST_FLAG /70/
DATA INFO_FLAG /71/
! User-open routines
INTEGER UFO_CREATE
EXTERNAL UFO_CREATE
    .
    .
    .
! Open the section file
STATUS = LIB$GET_LUN (INFO_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
SEC_MASK = SEC$M_WRT .OR. SEC$M_DZRO .OR. SEC$M_GBL
! (last address -- first address + length of last element + 511)/512
SEC_LEN = ( (%LOC(TERMINAL) - %LOC(DEVICE) + 6 + 511)/512 )
OPEN (UNIT=INFO_LUN,
2     FILE='INFO.TMP',
2     STATUS='NEW',
2     INITIALSIZE = SEC_LEN,
2     USEROPEN = UFO_CREATE)
! Free logical unit number and map section
CLOSE (INFO_LUN)
! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = %LOC (TERMINAL)
STATUS = SYS$CRMPSC (PASS_ADDR,      ! Address of section
2                    RET_ADDR,       ! Addresses mapped
2                    ,
2                    %VAL(SEC_MASK), ! Section mask
2                    'GLOBAL_SEC',   ! Section name
2                    ,,
2                    %VAL(SEC_CHAN), ! I/O channel
2                    ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
```

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Interprocess Communication Using Global Sections

```
! Create the subprocess
STATUS = SYS$CREPRC (,
2                    'GETDEVINF',     ! Image
2                    ,,,,,
2                    'GET_DEVICE',    ! Process name
2                    %VAL(4),,,)      ! Priority
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Write data to section
DEVICE = '$FLOPPY1'
! Get common event flag cluster and set flag
STATUS = SYS$ASCEFC (%VAL(REQUEST_FLAG),
2                    'CLUSTER',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$SETEF (%VAL(REQUEST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! When GETDEVINF has the information, INFO_FLAG is set
STATUS = SYS$WAITFR (%VAL(INFO_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
    .
    .
    .
```

GETDEVINF.FOR

```
! Define section flags
INCLUDE '($SECDEF)'
! Mask for section flags
INTEGER SEC_MASK
! Data for the section file
CHARACTER*12 DEVICE,
2            PROCESS
CHARACTER*6  TERMINAL
COMMON /DATA/ DEVICE,
2             PROCESS,
2             TERMINAL
! Location of data
INTEGER PASS_ADDR (2),
2       RET_ADDR (2)
! Two common event flags
INTEGER REQUEST_FLAG,
2       INFO_FLAG
DATA REQUEST_FLAG /70/
DATA INFO_FLAG /71/
    .
    .
    .
! Get common event flag cluster and wait
! for GBL1.FOR to set REQUEST_FLAG
STATUS = SYS$ASCEFC (%VAL(REQUEST_FLAG),
2                    'CLUSTER',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$WAITFR (%VAL(REQUEST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = %LOC (TERMINAL)
```

Example 5–1 Cont'd. on next page

Example 5–1 (Cont.) Interprocess Communication Using Global Sections

```
! Set write flag
SEC_MASK = SEC$M_WRT
! Map the section
STATUS = SYS$MGBLSC (PASS_ADDR,       ! Address of section
2                    RET_ADDR,        ! Address mapped
2                    ,
2                    %VAL(SEC_MASK),  ! Section mask
2                    'GLOBAL_SEC',,)  ! Section name
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Call GETDVI to get the process ID of the
! process that allocated the device, then
! call GETJPI to get the process name and terminal
! name associated with that process ID.
! Set PROCESS equal to the process name and
! set TERMINAL equal to the terminal name.
   .
   .
   .
! After information is in GLOBAL_SEC
STATUS = SYS$SETEF (%VAL(INFO_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

By default, a global section is deleted when no image is mapped to it. Such global sections are called temporary global sections. If you have PRMGBL privilege, you can create a permanent global section (set the SEC$V_PERM bit of the **flags** argument when you invoke SYS$CRMPSC). A permanent global section is not deleted until after it is marked for deletion with the SYS$DGBLSC system service (requires PRMGBL). Once a permanent section is marked for deletion, it is like a temporary section; when no image is mapped to it, the section is deleted.

5.3.4.3 VMS RMS Shared Files

VMS RMS allows concurrent access to a file. Shared files can be one of the following formats:

- Indexed files
- Relative files
- Sequential files with 512-byte fixed-length records

To coordinate access to a file, VMS RMS uses the lock manager. You can override the VMS RMS lock manager by controlling access yourself. Refer to Chapter 4 for more information on synchronization.

6 Security Features

With the VMS operating sytem, you can implement several security features to protect access to files and devices. The basis of the VMS security scheme is the *identifier*, a 32-bit binary value that represents a process to the system. An identifier can represent an individual user, a group of users, or some aspect of the environment in which a user is operating. A process is the *holder* of an identifier when that identifier can represent that process to the system.

6.1 Rights Database

The system *rights database* is an indexed file consisting of identifier and holder records. Those records define the identifiers and the holders of those identifiers on a system. When a process logs into the system, LOGINOUT creates a rights list for the process from the applicable entries in the rights database. Thus, a *process rights list* contains all the identifiers that the process holds. A process can be the holder of a number of identifiers. Each of those identifiers determines the identity and the access rights of the list holder. The process rights list becomes part of the process and is propagated to any created processes.

The entries in the rights list do not specifically grant access; instead, the VMS operating system uses the rights list to perform a protection check when the process attempts to access an object. VMS compares the identifiers in the rights list to the protection attributes of the object and grants or denies access to the object based on the comparison.

Access Control Lists

The VMS operating system also uses access control lists (ACL) in conjunction with the identifiers to control access to an object such as a file, device, or mailbox. An ACL consists of access control list entries (ACEs) that specify the type of access an identifier has to an object. When a process attempts to access an object with an associated ACL, the VMS operating system grants or denies access based on whether an exact match for the identifier in the ACL exists in the rights database.

6.2 System Services and Security

You use VMS system services to perform the following security tasks:

- Create and maintain a rights database
- Create and translate access control list entries
- Modify a process rights list
- Check access protection
- Provide a security erase pattern for disks
- Control magnetic tape access

Table 6–1 lists the system services related to system security.

Table 6–1 Security System Services

Service	Function
SYS$ADD_HOLDER	Adds holder record to rights database
SYS$ADD_IDENT	Adds identifier to rights database
SYS$ASCTOID	Translates identifier name to binary value
SYS$CHANGE_ACL	Creates or modifies an ACL
SYS$CHECK_ACCESS	Invokes system access protection check on behalf of another user
SYS$CHKPRO	Invokes system access protection check
SYS$CREATE_RDB	Initializes a rights database
SYS$ERAPAT	Generates a security erase pattern
SYS$FIND_HELD	Returns identifiers held by a holder in rights database
SYS$FIND_HOLDER	Returns holders of an identifier in rights database
SYS$FINISH_RDB	Deallocates record stream and clears context value when searching the rights database
SYS$FORMAT_ACL	Formats ACE into a text string
SYS$GRANTID	Adds identifier to process or system rights list
SYS$IDTOASC	Translates identifier value to its identifier name
SYS$MOD_HOLDER	Modifies holder record in rights database
SYS$MOD_IDENT	Modifies identifier record in rights database
SYS$MTACCESS	Controls magnetic tape access
SYS$PARSE_ACL	Converts text ACE into binary format
SYS$REM_HOLDER	Deletes holder record from identifier's list of holders in rights database
SYS$REM_IDENT	Deletes identifier and all holders of that identifier from rights database
SYS$REVOKID	Removes identifier from process or system rights list

The *VMS System Services Reference Manual* describes each major component of system security and how to use these system services to accomplish security tasks.

6.3 Privileged Images

In addition to using the system services, file security can also be provided by installing a program as a privileged image. When a program is installed as a privileged image, the program itself has specified privileges, thus eliminating the need for the user to have those privileges. To avoid security problems, you must prevent the privileged image from displaying traceback information; therefore, before installing the image, link it using the /NOTRACEBACK qualifier of the LINK command.

Perform the following steps to install a program as a privileged image:

1. Enter the DCL command SET PROCESS/PRIVILEGE=CMKRNL to give yourself CMKRNL privilege (required for use of the Install Utility).
2. Enter the INSTALL command at the $ prompt to invoke the interactive Install Utility.
3. When the *INSTALL>* prompt appears, enter the following command:

   ```
   INSTALL> CREATE file-specification /PRIVILEGED [=(priv,...)]
   ```

   The **priv** argument is a list of the privileges that the program requires. If only one privilege is specified, parentheses are not required.
4. Press the RETURN key. The Install Utility installs your program as a privileged image and reissues the *INSTALL>* prompt.
5. Enter the EXIT command to exit from the Install Utility.
6. Enter the DCL command SET PROCESS/PRIVILEGE=NOCMKRNL to remove the CMKRNL privilege.

The following statements install $DISK1:[INCOME]GET_STATS as a privileged image with the BYPASS privilege:

```
$ SET PROCESS/PRIVILEGE=CMKRNL
$ INSTALL
INSTALL> CREATE $DISK1:[INCOME]GET_STATS /PRIVILEGED=(BYPASS)
INSTALL> EXIT
$ SET PROCESS/PRIVILEGE=NOCMKRNL
```

A disk containing an installed image cannot be dismounted until the installed image is deleted. To delete an installed image, invoke the Install Utility and enter DELETE followed by the complete file specification of the image. Enter the EXIT subcommand to exit.

For more information about the Install Utility, see the *VMS Install Utility Manual*.

7 Input/Output Operations

The following techniques are available for completing I/O operations within a program:

- Program language I/O statements
- VMS Record Management Services (RMS) or RTL routines
- SYS$QIO and SYS$QIOW system services
- Non-DIGITAL-supplied device drivers to control the I/O to the device itself

The technique you select depends on the ease of use, speed, and level of control you want. The program language I/O statements have the least speed and level of control, but are the easiest to use. VMS RMS and RTL routines can perform most I/O operations for a high-level or assembly language program. System services can complete any I/O operation and can access devices not supported within VMS RMS. Writing a device driver provides the most control over I/O operations, but can be relatively difficult to implement.

This chapter describes the different levels of I/O programming and provides detailed examples of accomplishing common I/O tasks.

7.1 Choosing I/O Techniques

There are several types of I/O operations that can be performed within a program, including the following:

- Reading simple input from users and sending simple output to users
- Reading complex input from users and sending complex output to users
- Completing special I/O actions such as interrupts, controlling echo, handling unsolicited input, using the type-ahead buffer, using case conversion, and sending system broadcast messages
- Sending data to and from files
- Sending data to and from devices

7.1.1 Simple User I/O

To read simple input from a user or to send simple output to a user, use RTL routines. One RTL routine allows you to specify a prompt string to prompt for input from the current input device, defined by SYS$INPUT. Another RTL routine allows you to write a string to the current output device, defined by SYS$OUTPUT.

7.1.2 Complex User I/O

RTL routines provide an extensive number of screen management (SMG$) routines for reading multiple lines of input from users or for sending complex output to users. The SMG$ routines allow you to create and modify complicated displays that accept input and produce output.

7.1.3 Reading and Writing Data to Files

Programming language I/O statements can be the most effective for sending data to and from files. Program language I/O statements call VMS RMS routines to complete most file I/O. You can also use VMS RMS directly in your programs for accomplishing file I/O. File input/output operations are covered in Chapter 8.

7.1.4 Reading and Writing Data to Devices

To send data to and from devices, system services provide the most flexibility and control. You can use system services to access devices not supported by the programming language or by VMS RMS.

7.1.5 Broadcast Messages and Special I/O Actions

To complete special I/O actions, you can use SMG$ routines or the SYS$QIO or SYS$QIOW system services. For broadcast messages, use the SYS$BRKTHRU service.

7.2 Using SYS$INPUT and SYS$OUTPUT

Typically, you set up your program so that the user is the invoker. The user starts the program by entering a DCL command associated with the program or by using the RUN command.

7.2.1 Default Input and Output Devices

The user's input and output devices are defined by the logical names SYS$INPUT and SYS$OUTPUT, which are initially set to the values listed in Table 7–1.

Table 7–1 SYS$INPUT and SYS$OUTPUT Values

Logical Name	User Mode	Equivalence Device or File
SYS$INPUT	Interactive	Terminal on which user is logged in
	Batch job	Data lines following the invocation of the program
	Command procedure	Data lines following the invocation of the program

Table 7–1 (Cont.) SYS$INPUT and SYS$OUTPUT Values

Logical Name	User Mode	Equivalence Device or File
SYS$OUTPUT	Interactive	Terminal on which the user is logged in
	Batch job	Batch log file
	Command procedure	Terminal on which the user is logged in

Generally, use of SYS$INPUT and SYS$OUTPUT as the primary input and output devices is recommended. A user of the program can redefine SYS$INPUT and SYS$OUTPUT to redirect input and output as desired. For example, the interactive user might redefine SYS$OUTPUT as a file name to save output in a file rather than display it on the terminal.

7.2.2 Reading and Writing to Alternate Devices and External Files

Alternatively, you can design your program to read input from and write output to a file or a device other than the user's terminal. Files may be useful for writing lengthy amounts of data, for writing data that the user might want to save, and for writing data that can be reused as input. If you use files or devices other than SYS$INPUT and SYS$OUTPUT, you should provide the names of the files or devices (best form is to use logical names) and any conventions for their use. You can specify such information by having the program write it to the terminal, by creating a help file, or by providing user documentation.

7.3 Working with Simple User I/O

Usually, you can request information from, or write information to, the user with little regard for formatting. For such simple I/O, use LIB$GET_INPUT and LIB$PUT_OUTPUT or the I/O statements for your programming language.

To provide complex screen displays for input or output, use the screen management facility described in Section 7.4.

7.3.1 Default Devices for Simple I/O

LIB$GET_INPUT and LIB$PUT_OUTPUT read from SYS$INPUT and write to SYS$OUTPUT. The logical names SYS$INPUT and SYS$OUTPUT are implicit to the routines; you need only call the routine to access the I/O unit (device or file) associated with SYS$INPUT and SYS$OUTPUT. You cannot use these routines to access an I/O unit other than the one associated with SYS$INPUT or SYS$OUTPUT.

If more than one person is working on a program, you should generate logical unit numbers with LIB$GET_LUN, rather than choose your own values, to ensure that the number is unique among all logical unit numbers used in the program.

7.3.2 Getting a Line of Input

A read operation transfers one record from the input unit to a variable or variables of your choice. On a terminal, the user ends a record by pressing a terminator. The terminators are the ASCII characters NUL through US (0 through 31) except for LF, VT, FF, TAB, and BS. The usual terminator is CR, generated by pressing the RETURN key.

If you are reading character data, LIB$GET_INPUT is a simple way of prompting for and reading the data. If you are reading noncharacter data, programming language I/O statements are preferable since they allow you to translate the data to a format of your choice.

For example, VAX FORTRAN I/O offers the ACCEPT statement, which reads data from SYS$INPUT, and the READ statement, which reads data from an I/O unit of your choice.

Make sure the variables that you specify can hold the largest number of characters the user of your program might enter, unless you deliberately want to truncate the input. Overflowing the input variable using LIB$GET_INPUT causes the fatal error LIB$_INPSTRTRU (defined in $LIBDEF); overflowing the input variable using language I/O statements may not necessarily cause an error but does truncate your data.

LIB$GET_INPUT places the characters read in a variable of your choice. You must define the variable type as character. Optionally, LIB$GET_INPUT places the number of characters read in another variable of your choice. On terminal input, LIB$GET_INPUT optionally writes a prompt before reading the input. The prompt is suppressed automatically for a nonterminal operation.

Example 7–1 reads a line of input using LIB$GET_INPUT:

Example 7–1 Reading a Line of Data

```
INTEGER*4     STATUS,
2             LIB$GET_INPUT
INTEGER*2     INPUT_SIZE
CHARACTER*512 INPUT
STATUS = LIB$GET_INPUT (INPUT,           ! Input value
2                       'Input value: ', ! Prompt (optional)
2                       INPUT_SIZE)      ! Input size (optional)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

In further references to input character data, you should specify the appropriate substring or sequence of array elements rather than the entire character variable. Using the preceding example, if you read characters into a character string variable named INPUT and store the number of characters read in a variable named INPUT_SIZE, you should refer to INPUT (1:INPUT_SIZE) rather than INPUT.

7.3.3 Getting Several Lines of Input

The usual technique for getting a variable number of input records—either values for which you are prompting or data records from a file—is to read and process records until end-of-file. End-of-file means one of the following:

- Terminal—The user has pressed CTRL/Z. To ensure that the convention is followed, you might first write a message telling the user to press CTRL/Z to terminate the input sequence.
- Command procedure—The end of a sequence of data lines has been reached.
- File—The end of an actual file has been reached.

Process the records in a loop (one record per iteration) and terminate the loop on end-of-file. LIB$GET_INPUT returns the error RMS$_EOF (defined in $RMSDEF) when end-of-file occurs.

Example 7–2 uses a VAX FORTRAN READ statement in a loop to read a sequence of integers from SYS$INPUT:

Example 7–2 Reading a Varying Number of Input Records

```
! Return status and error codes
INTEGER    STATUS,
2          IOSTAT,
3          STATUS_OK,
4          IOSTAT_OK
PARAMETER (STATUS_OK = 1,
2           IO_OK = 0)
INCLUDE    '($FORDEF)'
! Data record read on each iteration
INTEGER    INPUT_NUMBER
! Accumulated data records
INTEGER    STORAGE_COUNT,
2          STORAGE_MAX
PARAMETER (STORAGE_MAX = 255)
INTEGER     STORAGE_NUMBER (STORAGE_MAX)
! Write instructions to interactive user
TYPE *,
2 'Enter values below. Press CTRL/Z when done.'
! Get first input value
WRITE (UNIT=*,
2       FMT='(A,$)') ' Input value: '
READ (UNIT=*,
2      IOSTAT=IOSTAT,
2      FMT='(BN,I)') INPUT_NUMBER
IF (IOSTAT .EQ. IO_OK) THEN
  STATUS = STATUS_OK
ELSE
  CALL ERRSNS (,,,,STATUS)
END IF
```

Example 7–2 Cont'd. on next page

Example 7–2 (Cont.) Reading a Varying Number of Input Records

```
! Process each input value until end-of-file
DO WHILE ((STATUS .NE. FOR$_ENDDURREA) .AND.
          (STORAGE_COUNT .LT. STORAGE_MAX))
  ! Keep repeating on conversion error
  DO WHILE (STATUS .EQ. FOR$_INPCONERR)
    WRITE (UNIT=*,
2          FMT='(A,$)') ' Try again: '
    READ (UNIT=*,
2         IOSTAT=IOSTAT,
2         FMT='(BN,I)') INPUT_NUMBER
    IF (IOSTAT .EQ. IO_OK) THEN
      STATUS = STATUS_OK
    ELSE
      CALL ERRSNS (,,,,STATUS)
    END IF
  END DO
  ! Continue if end-of-file not entered
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
    ! Status check on last read
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Store input numbers in input array
    STORAGE_COUNT = STORAGE_COUNT + 1
    STORAGE_NUMBER (STORAGE_COUNT) = INPUT_NUMBER
    ! Get next input value
    WRITE (UNIT=*,
2          FMT='(A,$)') ' Input value: '
    READ (UNIT=*,
2         IOSTAT=IOSTAT,
2         FMT='(BN,I)') INPUT_NUMBER
    IF (IOSTAT .EQ. IO_OK) THEN
      STATUS = STATUS_OK
    ELSE
      CALL ERRSNS (,,,,STATUS)
    END IF
  END IF
END DO
```

7.3.4 Writing Simple Output

You can use LIB$PUT_OUTPUT to write character data. If you are writing noncharacter data, programming language I/O statements are preferable, since they allow you to translate the data to a format of your choice.

LIB$PUT_OUTPUT writes one record of output to SYS$OUTPUT. Typically, you should avoid writing records that exceed the device width. The width of a terminal is 80 or 132 characters, depending on the setting of the physical characteristics of the device. The width of a line printer is 132 characters. If your output record exceeds the width of the device, the excess characters are either truncated or wrapped to the next line, depending on the setting of the physical characteristics.

You must define a value (a variable, constant, or expression) to be written. The value must be expressed in characters. You should specify the exact number of characters being written and not include the trailing portion of a variable.

The following example writes a character expression to SYS$OUTPUT:

```
INTEGER*4     STATUS,
2             LIB$PUT_OUTPUT
CHARACTER*40 ANSWER
INTEGER*4     ANSWER_SIZE
    .
    .
    .
STATUS = LIB$PUT_OUTPUT ('Answer: ' // ANSWER (1:ANSWER_SIZE))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4 Working with Complex User I/O

The SMG$ run-time library routines provide a simple, device-independent interface for managing the appearance of the terminal screen. The SMG$ routines are primarily for use with video terminals; however, they can be used with files or hardcopy terminals.

To use the Screen Management Facility for output, do the following:

1. Create a pasteboard—A pasteboard is a logical, two-dimensional area on which you place virtual displays. Use the SMG$CREATE_PASTEBOARD routine to create a pasteboard, and associate it with a physical device. When you refer to the pasteboard, SMG performs the necessary I/O operation to the device.
2. Create a virtual display—A virtual display is a logical, two-dimensional area in which you place the information to be displayed. Use the SMG$CREATE_VIRTUAL_DISPLAY routine to create a virtual display.
3. Paste virtual displays to the pasteboard—To make a virtual display visible, map (or paste) it to the pasteboard using the SMG$PASTE_VIRTUAL_DISPLAY routine. You can reference a virtual display regardless of whether that display is currently pasted to a pasteboard.
4. Create a viewport for a virtual display—A viewport is a rectangular viewing area that can be moved around on a buffer to view different pieces of the buffer. The viewport is associated with a virtual display.

Example 7–3 associates a pasteboard with the terminal, creates a virtual display the size of the terminal screen, and pastes the display to the pasteboard. When text is written to the virtual display, it appears on the terminal screen.

Example 7–3 Associating a Pasteboard with a Terminal

```
! Screen management control structures
INTEGER*4 PBID,    ! Pasteboard ID
2         VDID,    ! Virtual display ID
2         ROWS,    ! Rows on screen
2         COLS     ! Columns on screen
! Status variable and routines called as functions
INTEGER*4 STATUS,
2         SMG$CREATE_PASTEBOARD,
2         SMG$CREATE_VIRTUAL_DISPLAY,
2         SMG$PASTE_VIRTUAL_DISPLAY
! Set up SYS$OUTPUT for screen management
! and get the number of rows and columns on the screen
STATUS = SMG$CREATE_PASTEBOARD (PBID,    ! Return value
2                              'SYS$OUTPUT',
2                              ROWS,     ! Return value
2                              COLUMNS) ! Return value
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create virtual display that pastes to the full screen size
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (ROWS,
2                                   COLUMNS,
2                                   VDID) ! Return value
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Paste virtual display to pasteboard
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (VDID,
2                                  PBID,
2                                  1, ! Starting at row 1 and
2                                  1) ! column 1 of the screen
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
```

To use the SMG$ routines for input, you associate a virtual keyboard with a physical device or file using the SMG$CREATE_VIRTUAL_KEYBOARD routine. The SMG$ input routines can be used alone or with the output routines. This section assumes that you are using the input routines with the output routines. Section 7.5 describes how to use the input routines without the output routines.

The Screen Management Facility keeps an internal representation of the screen contents; therefore, it is important that you do not mix SMG$ routines with other forms of terminal I/O. The following subsections contain guidelines for using most of the SMG$ routines; for more details, see the *VMS Run-Time Library Routines Volume.*

7.4.1 Pasteboards

Use the SMG$CREATE_PASTEBOARD routine to create a pasteboard and associate it with a physical device. SMG$CREATE_PASTEBOARD returns a unique pasteboard identification number; use that number to refer to the pasteboard in subsequent calls to SMG$ routines. After associating a pasteboard with a device, your program references only the pasteboard. The Screen Management Facility performs all necessary operations between the pasteboard and the physical device.

7.4.1.1 Creating a Pasteboard

When you create a pasteboard, the Screen Management Facility clears the screen by default. To clear the screen yourself, invoke the SMG$ERASE_PASTEBOARD routine. Any virtual displays associated with the pasteboard are removed from the screen, but their contents in memory are not affected. The following example erases the screen:

```
STATUS = SMG$ERASE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.1.2 Deleting a Pasteboard

Invoking SMG$DELETE_PASTEBOARD deletes a pasteboard, making the screen unavailable for further pasting. The optional second argument of the SMG$DELETE_PASTEBOARD routine allows you to indicate whether the routine clears the screen (the default) or leaves it as is. The following example deletes a pasteboard and clears the screen:

```
STATUS = SMG$DELETE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

By default, the screen is erased when you create a pasteboard. Generally, you should erase the screen at the end of a session.

7.4.1.3 Setting Screen Dimensions and Background Color

The routine SMG$CHANGE_PBD_CHARACTERISTICS sets the dimensions of the screen and its background color. You can also use this routine to retrieve dimensions and background color. To get more detailed information about the physical device, use the SMG$GET_PASTEBOARD_ATTRIBUTES routine. The following example changes the screen width to 132 and the background to white, then restores the original width and background before exiting:

Example 7–4 Modifying the Screen Dimensions and Background Color

```
         .
         .
         .
INTEGER*4 WIDTH,
2         COLOR
INCLUDE   '($SMGDEF)'
```

Example 7–4 Cont'd. on next page

Example 7–4 (Cont.) Modifying the Screen Dimensions and Background Color

```
! Get current width and background color
STATUS = SMG$CHANGE_PBD_CHARACTERISTICS (PBID,,
2                                        WIDTH,,,,
2                                        COLOR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Change width and background color
STATUS = SMG$CHANGE_PBD_CHARACTERISTICS (PBID,
2                                        132,,,,
2                                        SMG$C_COLOR_WHITE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
! Restore width and background color
STATUS = SMG$CHANGE_PBD_CHARACTERISTICS (PBID,
2                                        WIDTH,,,,
2                                        COLOR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.2 Virtual Displays

You write to virtual displays, which are logically configured as rectangles, using the SMG$ routines. The dimensions of a virtual display are designated vertically as so many rows and horizontally as so many columns. A position in a virtual display is designated by naming a row and a column. Row and column numbers begin at 1.

7.4.2.1 Creating a Virtual Display

Use the SMG$CREATE_VIRTUAL_DISPLAY routine to create a virtual display. SMG$CREATE_VIRTUAL_DISPLAY returns a unique virtual display identification number; use that number to refer to the virtual display.

Optionally, you can use the fifth argument of SMG$CREATE_VIRTUAL_DISPLAY to specify one or more of the following video attributes: blinking, bolding, reversing background, and underlining. All characters written to that display will have the specified attribute unless you indicate otherwise when writing text to the display. The following example makes everything written to the display HEADER_VDID appear bolded by default:

```
INCLUDE '($SMGDEF)'
   .
   .
   .
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (1,   ! Rows
2                                    80,  ! Columns
2                                    HEADER_VDID,,
2                                    SMG$M_BOLD)
```

You can border a virtual display by specifying the fourth argument when you invoke SMG$CREATE_VIRTUAL_DISPLAY. You can label the border with the routine SMG$LABEL_BORDER. If you use a border, you must leave room for it: a border requires two rows and two columns more than the size of the display. The following example places a labeled border around the STATS_VDID display. As pasted, the border occupies rows 2 and 13 and columns 1 and 57.

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (10,  ! Rows
2                                    55,  ! Columns
2                                    STATS_VDID,
2                                    SMG$M_BORDER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$LABEL_BORDER (STATS_VDID,
2                          'statistics')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                   PBID,
2                                   3,  ! Row
2                                   2)  ! Column
```

7.4.2.2 Pasting Virtual Displays

To make a virtual display visible, paste it to a pasteboard using the SMG$PASTE_VIRTUAL_DISPLAY routine. You position the virtual display by specifying which row and column of the pasteboard should contain the upper lefthand corner of the display. Example 7–5 defines two virtual displays and pastes them to one pasteboard.

Example 7–5 Defining and Pasting Virtual Displays

```
     .
     .
     .
INCLUDE '($SMGDEF)'
INTEGER*4 PBID,
2         HEADER_VDID,
2         STATS_VDID
INTEGER*4 STATUS,
2         SMG$CREATE_PASTEBOARD,
2         SMG$CREATE_VIRTUAL_DISPLAY,
2         SMG$PASTE_VIRTUAL_DISPLAY
! Create pasteboard for SYS$OUTPUT
STATUS = SMG$CREATE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Header pastes to first rows of screen
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (3,            ! Rows
2                                    80,           ! Columns
2                                    HEADER_VDID,  ! Name
2                                    SMG$M_BORDER) ! Border
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (HEADER_VDID,
2                                   PBID,
2                                   1,             ! Row
2                                   1)             ! Column
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

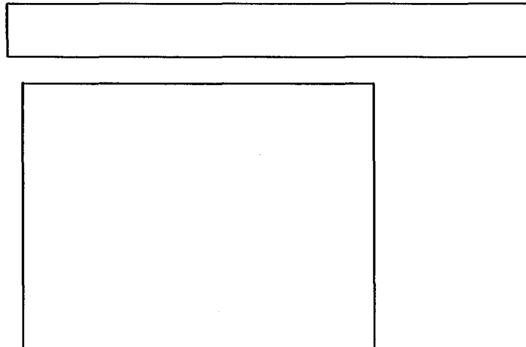
Example 7–5 Cont'd. on next page

Example 7–5 (Cont.) Defining and Pasting Virtual Displays

```
! Statistics area pastes to rows 5 through 15,
! columns 2 through 56
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (10,            ! Rows
2                                    55,            ! Columns
2                                    STATS_VDID,    ! Name
2                                    SMG$M_BORDER)  ! Border
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                   PBID,
2                                   5,              ! Row
2                                   2)              ! Column
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
```

Figure 7–1 shows the resultant screen.

Figure 7–1 Defining and Pasting Virtual Displays



ZK-2044-84

You can paste a single display to any number of pasteboards. Any time you change the display, all pasteboards containing the display are automatically updated.

A pasteboard can hold any number of virtual displays. You can paste virtual displays over one another to any depth, occluding the displays underneath. The displays underneath are only occluded to the extent that they are covered; that is, the parts not occluded remain visible on the screen. (In the first figure of Section 7.4.2.3, displays 1 and 2 are partially occluded.) When you unpaste a virtual display that occludes another virtual display, the occluded part of the underneath display becomes visible again.

You can find out if a display is occluded with the routine SMG$CHECK_FOR_OCCLUSION. The following example pastes a 2-row summary display over the last two rows of the statistics display, if the statistics display is not already occluded. If the statistics display is occluded, the example assumes that it is occluded by the summary display and unpastes the summary display, making the last two rows of the statistics display visible again.

```
  STATUS = SMG$CHECK_FOR_OCCLUSION (STATS_VDID,
2                                   PBID,
2                                   OCCLUDE_STATE)
! OCCLUDE_STATE must be defined as INTEGER*4
  IF (OCCLUDE_STATE) THEN
    STATUS = SMG$UNPASTE_VIRTUAL_DISPLAY (SUM_VDID,
2                                         PBID)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ELSE
    STATUS = SMG$PASTE_VIRTUAL_DISPLAY (SUM_VDID,
2                                       PBID,
2                                       11,
2                                       2)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
```

7.4.2.3 Rearranging Virtual Displays

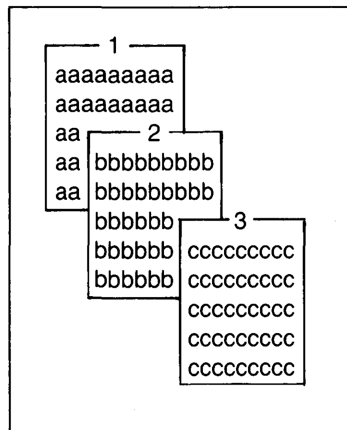
Pasted displays can be rearranged by moving or repasting.

- Moving—To move a display, use the SMG$MOVE_VIRTUAL_DISPLAY routine. The following example moves display 2. Figure 7–2 shows the screen before and after the statement executes.

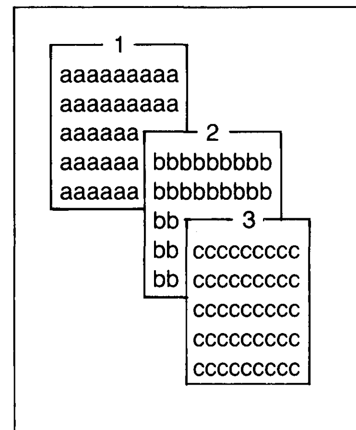
```
STATUS = SMG$MOVE_VIRTUAL_DISPLAY (VDID,
2                                  PBID,
2                                  5,
2                                  10)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Figure 7–2 Moving a Virtual Display

Before Moving Display 2



After Moving Display 2

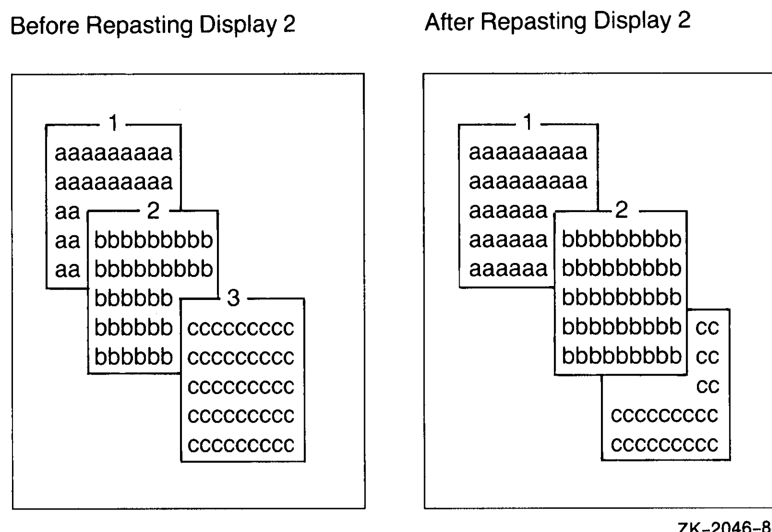


ZK-2045-84

- Repasting—To repaste a display, use the SMG$REPASTE_VIRTUAL_DISPLAY routine. The following example repastes display 2. Figure 7–3 shows the screen before and after the statement executes.

```
STATUS = SMG$REPASTE_VIRTUAL_DISPLAY (VDID,
2                                     PBID,
2                                     4,
2                                     4)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Figure 7–3 Repasting a Virtual Display



You can obtain the pasting order of the virtual displays using SMG$LIST_PASTING_ORDER. This routine returns the identifiers of all the virtual displays pasted to a specified pasteboard.

7.4.2.4 Removing Virtual Displays

You can remove a virtual display from a pasteboard in a number of different ways:

- Erase a virtual display—Invoking SMG$UNPASTE_VIRTUAL_DISPLAY erases a virtual display from the screen but retains its contents in memory. The following example erases the statistics display:

```
STATUS = SMG$UNPASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                     PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- Delete a virtual display—Invoking SMG$DELETE_VIRTUAL_DISPLAY removes a virtual display from the screen and removes its contents from memory. The following example deletes the statistics display:

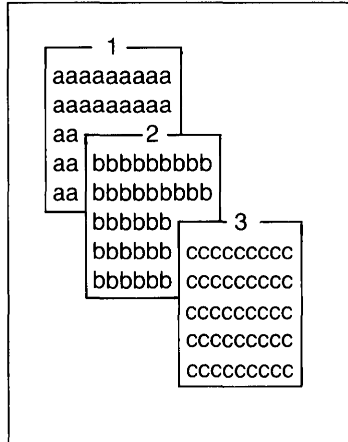
```
STATUS = SMG$DELETE_VIRTUAL_DISPLAY (STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- Delete a number of virtual displays—Invoking SMG$POP_VIRTUAL_DISPLAY removes a specified virtual display and any virtual displays pasted after that display from the screen and removes the contents of those displays from memory. The following example "pops" display 2. Figure 7–4 shows the screen before and after popping. (Note that display 3 is not deleted because it is occluding display 2, but because it was pasted after display 2.)

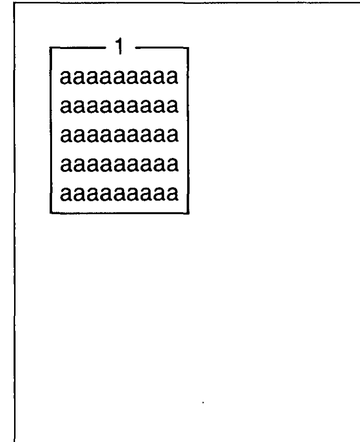
```
STATUS = SMG$POP_VIRTUAL_DISPLAY (STATS_VDID,
2                                 PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Figure 7–4 Popping a Virtual Display

Before Popping Display 2



After Popping Display 2



ZK-2047-84

7.4.2.5 Modifying a Virtual Display

The screen management facility provides several routines for modifying the characteristics of an existing virtual display:

- SMG$CHANGE_VIRTUAL_DISPLAY—Changes the size, video attributes, or border of a display
- SMG$CHANGE_RENDITION—Changes the video attributes of a portion of a display
- SMG$MOVE_TEXT—Moves text from one virtual display to another

The following example uses SMG$CHANGE_VIRTUAL_DISPLAY to change the size of the WHOOPS display to five rows and seven columns and to turn off all of the display's video attributes. If you decrease the size of a display that is on the screen, any characters in the excess area are removed from the screen.

```
STATUS = SMG$CHANGE_VIRTUAL_DISPLAY (WHOOPS_VDID,
2                                    5,  ! Rows
2                                    7,, ! Columns
2                                    0)  ! Video attributes off
```

The following example uses SMG$CHANGE_RENDITION to direct attention to the first 20 columns of the statistics display by setting the reverse video attribute to the complement of the display's default setting for that attribute:

```
STATUS = SMG$CHANGE_RENDITION (STATS_VDID,
2                             1,              ! Row
2                             1,              ! Column
2                             10,             ! Number of rows
2                             20,             ! Number of columns
2                             ,               ! Video-set argument
2                             SMG$M_REVERSE)  ! Video-comp argument
2
```

SMG$CHANGE_RENDITION uses three sets of video attributes to determine the attributes to apply to the specified portion of the display: (1) the display's default video attributes, (2) the attributes specified by the **rendition-set** argument of SMG$CHANGE_RENDITION, and the (3) attributes specified by the **rendition-complement** argument of SMG$CHANGE_RENDITION. Table 7–2 shows the result of each possible combination:

Table 7–2 Setting Video Attributes

rendition-set	rendition-complement	Result
off	off	Uses display default
on	off	Sets attribute
off	on	Uses the complement of display default
on	on	Clears attribute

In the previous example, the reverse video attribute is set in the **rendition-complement** argument but not in the **rendition-set** argument, thus specifying that SMG$CHANGE_RENDITION use the complement of the display's default setting to ensure that the selected portion of the display is easily seen.

Note that the resulting attributes are based on the display's default attributes, not its current attributes. If you use SMG$ routines that explicitly set video attributes, the current attributes of the display may not match its default attributes.

7.4.2.6 Using Spawned Subprocesses

You can create a spawned subprocess directly with an SMG$ routine to allow execution of a DCL command from an application. Only one spawned subprocess is allowed per virtual display. Use the following routines to work with subprocesses:

- SMG$CREATE_SUBPROCESS—Creates a DCL spawned subprocess and associates it with a virtual display.
- SMG$EXECUTE_COMMAND—Allows execution of a specified command in the created spawned subprocess. There are some restrictions in specifying the command as follows:
  - SPAWN, GOTO, or LOGOUT cannot be used and would result in unpredictable results.
  - Single-character commands such as CTRL/C have no effect. You can signal an end-of-file (that is, CTRL/Z) command by setting the **flags**.

  - A dollar sign must be specified as the first character of any DCL command.
- SMG$DELETE_SUBPROCESS—Deletes the subprocess created by SMG$CREATE_SUBPROCESS.

7.4.3 Viewports

Viewports allow you to view different pieces of a virtual display by moving a rectangular area around on the virtual display. Only one viewport is allowed for each virtual display. Once you have associated a viewport with a virtual display, the only part of the virtual display that is viewable is contained in the viewport.

The SMG$ routines for working with viewports include the following:

- SMG$CREATE_VIEWPORT—Creates a viewport and associates it with a virtual display. You must create the virtual display first. To view the viewport, you must paste the virtual display first with SMG$PASTE_VIRTUAL_DISPLAY.
- SMG$SCROLL_VIEWPORT—Scrolls the viewport within the virtual display. If you try to move the viewport outside of the virtual display, it is truncated to stay within the virtual display. This routine allows you to specify the direction and extent of the scroll.
- SMG$CHANGE_VIEWPORT—Moves the viewport to a new starting location and changes the size of the viewport.
- SMG$DELETE_VIEWPORT—Deletes the viewport and dissociates it from the virtual display. The viewport is automatically unpasted. The virtual display associated with the viewport remains intact. You can unpaste a viewport without deleting it, using SMG$UNPASTE_VIRTUAL_DISPLAY.

7.4.4 Writing Text to Virtual Display

The SMG$ output routines allow you to write text to displays and to delete or modify the existing text of a display. Remember that changes to a virtual display are visible only if the virtual display is pasted to a pasteboard.

7.4.4.1 Positioning the Cursor

Each virtual display has its own logical cursor position. You can control the position of the cursor in a virtual display with the following routines:

- SMG$HOME_CURSOR—Moves the cursor to a corner of the virtual display. The default corner is the upper left corner, that is, row 1 column 1 of the display.
- SMG$SET_CURSOR_ABS—Moves the cursor to a specified row and column.
- SMG$SET_CURSOR_REL—Moves the cursor to offsets from the current cursor position. A negative value means up (rows) or left (columns). Zero means no movement.

In addition, many routines permit you to specify a starting location other than the current cursor position for the operation.

The routine SMG$RETURN_CURSOR_POS returns the row and column of the current cursor position within a virtual display. You do not have to write special code to track the cursor position.

Typically, the physical cursor is at the logical cursor position of the most recently written-to display. If necessary, you can use the SMG$SET_PHYSICAL_CURSOR routine to set the physical cursor location.

7.4.4.2 Writing Data Character by Character

If you are writing character by character (see Section 7.4.4.3 for line-oriented output), there are three routines to use:

- SMG$DRAW_CHAR—Puts one character on the screen at a specified position. It does not change the cursor position.
- SMG$PUT_CHARS—Puts several characters on the screen at a specified position with the option of one video attribute.
- SMG$PUT_CHARS_MULTI—Puts several characters on the screen at a specified position, with multiple video attributes.

These routines are simple and precise. They place exactly the specified characters on the screen, starting at a specified position in a virtual display. Anything currently in the positions written-to is overwritten; no other positions on the screen are affected. Convert numeric data to character data with language I/O statements before invoking SMG$PUT_CHARS.

The following example converts an integer to a character string and places it at a designated position in a virtual display:

```
CHARACTER*4 HOUSE_NO_STRING
INTEGER*4   HOUSE_NO,
2           LINE_NO,
2           STATS_VDID
    .
    .
    .
WRITE (UNIT=HOUSE_NO_STRING,
2     FMT='(I4)') HOUSE_NO
STATUS = SMG$PUT_CHARS (STATS_VDID,
2                       HOUSE_NO_STRING,
2                       LINE_NO,   ! Row
2                       1)         ! Column
```

Note that the converted integer is right-justified from column 4 because the format specification is I4 and the full character string is written. To left-justify a converted number, you must locate the first nonblank character and write a substring starting with that character and ending with the last character.

Inserting and Overwriting Text

To insert characters rather than overwrite the current contents of the screen, use the routine SMG$INSERT_CHARS. Existing characters at the location written to are shifted to the right. Characters pushed out of the display are truncated; no wrapping occurs and the cursor remains at the end of the last character inserted.

Specifying Double-Width Characters

In addition to the aforementioned routines, you can use SMG$PUT_CHARS_WIDE to write characters to the screen in double width or SMG$PUT_CHARS_HIGHWIDE to write characters to the screen in double height and double width. When you use these routines, you must allot two spaces for each double-width character on the line and two lines for each line of double-height characters. You cannot mix single and double-size characters on a line.

All the character routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the characters being written. SMG$PUT_CHARS_MULTI allows you to specify more than one video attribute at a time. The explanation of the SMG$CHANGE_RENDITION routine in Section 7.4.2.5 discusses how to use **rendition-set** and **rendition-complement** arguments.

7.4.4.3 Writing Data Line by Line

The routines SMGPUT_LINE, SMGPUT_LINE_MULTI, SMG$PUT_WITH_SCROLL write lines to virtual displays one line after another. If the display area is full, it is scrolled. You do not have to keep track of which line you are on. All routines permit you to scroll forward (up); SMG$PUT_WITH_SCROLL and SMG$PUT_LINE_MULTI permit you to scroll backward (down) as well. SMG$PUT_LINE permits other than single spacing.

Example 7–6 writes lines from a buffer to a display area. The output is scrolled forward if the buffer contains more lines than the display area.

Example 7–6 Scrolling Forward Through a Display

```
INTEGER*4     BUFF_COUNT,
2             BUFF_SIZE (4096)
CHARACTER*512 BUFF (4096)
   .
   .
   .
DO I = 1, BUFF_COUNT
  STATUS = SMG$PUT_WITH_SCROLL (VDID,
2                                 BUFF (I) (1:BUFF_SIZE (I)))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
```

Example 7–7 scrolls the output backward.

Example 7–7 Scrolling Backward Through the Display

```
DO I = BUFF_COUNT, 1, -1
  STATUS = SMG$PUT_WITH_SCROLL (VDID,
2                              BUFF (I) (1:BUFF_SIZE (I)),
2                              SMG$M_DOWN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
```

Cursor Movement and Scrolling

To maintain precise control over cursor movement and scrolling, you can write with SMG$PUT_CHARS and scroll explicitly with SMG$SCROLL_DISPLAY_AREA. SMG$PUT_CHARS leaves the cursor after the last character written and does not force scrolling; SMG$SCROLL_DISPLAY_AREA scrolls the current contents of the display forward, backward, or sideways without writing to the display. To restrict the scrolling region to a portion of the display area, use the SMG$SET_DISPLAY_SCROLL_REGION routine.

Inserting and Overwriting Text

To insert text rather than overwrite the current contents of the screen, use the routine SMG$INSERT_LINE. Existing lines are shifted up or down to open space for the new text. If the text is longer than a single line, you can specify whether or not you want the excess characters to be truncated or wrapped.

Using Double-Width Characters

In addition, you can use SMG$PUT_LINE_WIDE to write a line of text to the screen using double-width characters. You must allot two spaces for each double-width character on the line. You cannot mix single- and double-width characters on a line.

Specifying Special Video Attributes

All line routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the text being written. SMG$PUT_LINE_MULTI allows you to specify more than one video attribute for the text. The explanation of the SMG$CHANGE_RENDITION routine in Section 7.4.2.5 discusses how to use the **rendition-set** and **rendition-complement** arguments.

7.4.4.4 Drawing Lines

The routine SMG$DRAW_LINE draws solid lines on the screen. Appropriate corner and crossing marks are drawn when lines join or intersect. You can also use the routine SMG$DRAW_RECTANGLE to draw a solid rectangle. Suppose that you want to draw an object such as that shown in Figure 7–5 in the statistics display area (an area of 10 rows by 55 columns).

Figure 7–5 Statistics Display

ZK-2048-84

Example 7–8 shows how you can create a statistics display using SMG$DRAW_LINE and SMG$DRAW_RECTANGLE.

Example 7–8 Creating a Statistics Display

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (10,
2                                    55,
2                                    STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Draw rectangle with upper left corner at row 1 column 1
! and lower right corner at row 10 column 55
STATUS =SMG$DRAW_RECTANGLE (STATS_VDID,
2                           1, 1,
2                           10, 55)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Draw vertical lines at columns 11, 21, and 31
DO I = 11, 31, 10
  STATUS = SMG$DRAW_LINE (STATS_VDID,
2                         1, I,
2                         10, I)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
! Draw horizontal line at row 3
STATUS = SMG$DRAW_LINE (STATS_VDID,
2                       3, 1,
2                       3, 55)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                   PBID,
2                                   3,
2                                   2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.4.5 Deleting Text

The following routines erase specified characters leaving the rest of the screen intact:

- SMG$ERASE_CHARS—Erases specified characters on one line.
- SMG$ERASE_LINE—Erases the characters on one line starting from a specified position.
- SMG$ERASE_DISPLAY—Erases specified characters on one or more lines.

- SMG$ERASE_COLUMN—Erases a column from the specified row to the end of the column from the virtual display.

The following routines perform delete operations. In a delete operation, characters following the deleted characters are shifted into the empty space.

- SMG$DELETE_CHARS—Deletes specified characters on one line. Any characters to the right of the deleted characters are shifted left.
- SMG$DELETE_LINE—Deletes one or more full lines. Any remaining lines in the display are scrolled up to fill the empty space.

The following example erases the remaining characters on the line whose line number is specified by LINE_NO, starting at the column specified by COLUMN_NO:

```
STATUS = SMG$ERASE_LINE (STATS_VDID,
2                        LINE_NO,
2                        COLUMN_NO)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.5 Using Menus

You can use SMG$ routines to set up menus to read user input. The type of menus you can create include the following:

- Block menu—Selections are in matrix format. This is the type of menu often used.
- Vertical menu—Each selection is on its own line.
- Horizontal menu—All selections are on one line.

Menus are associated with a virtual display and only one menu can be used for each virtual display.

The menu routines include the following:

- SMG$CREATE_MENU—Creates a menu associated with a virtual display. This routine allows you to specify the type of menu, the position the menu is displayed, the format of the menu (single or double spaced), and video attributes.
- SMG$SELECT_FROM_MENU—Sets up menu selection capability. You can specify a default menu selection (which is shown in reverse video), whether on-line HELP is available, a maximum time limit for making a menu selection, a key indicating read termination, whether to send the text of the menu item selected to a string, and a video attribute.
- SMG$DELETE_MENU—Discontinues access to the menu and erases it.

When you are using menus, no other output should be sent to the menu area; otherwise unpredictable results may occur.

The default SMG$SELECT_FROM_MENU allows specific operations such as use of the arrow keys to move up and down the menu selections, keys to make a menu selection, ability to select more than one item at a time, ability to reselect an item already selected, and the key sequence to invoke on-line HELP. By using the **flags** argument to modify this operation, you have the option of disallowing reselection of a menu item and allowing any key pressed to select an item.

7.4.6 Reading Data

You can read text from a virtual display (SMG$READ_FROM_DISPLAY) or from a virtual keyboard (SMG$READ_STRING or SMG$READ_COMPOSED_LINE). The two routines for virtual keyboard input are known as the SMG$ input routines. SMG$READ_FROM_DISPLAY is not a true input routine because it reads text from the virtual display rather than from a user.

The SMG$ input routines can be used alone or with the SMG$ output routines. This section assumes that you are using the input routines with the output routines. Section 7.5 describes how to use the input routines without the output routines.

When using the SMG$ input routines with the SMG$ output routines, always specify the optional **vdid** argument of the input routine, which specifies the virtual display in which the input is to occur. The specified virtual display must be pasted to the device associated with the virtual keyboard that is specified as the first argument of the input routine. The display must be pasted in column 1, cannot be occluded, and cannot have any other display to its right; input begins at the current cursor position, but the cursor must be in column 1.

7.4.6.1 Reading from a Display

You can read the contents of the screen using the routine SMG$READ_FROM_DISPLAY. By default, the read operation reads all of the characters from the current cursor position to the end of that line. The **row** argument of SMG$READ_FROM_DISPLAY allows you to choose the starting point of the read operation, that is, the contents of the specified row to the rightmost column in that row.

If the **terminator-string** argument is specified, SMG$READ_FROM_DISPLAY searches backward from the current cursor position and reads the line beginning at the first *terminator* encountered (or at the beginning of the line). A terminator is a character string. You must calculate the length of the character string read operation yourself.

The following example reads the current contents of the first line in the STATS_VDID display. To ensure that the display is up to date, SMG$READ_FROM_DISPLAY automatically invokes SMG$FLUSH_BUFFER before reading from the display.

```
CHARACTER*4 STRING
INTEGER*4   SIZE
    .
    .
    .
STATUS = SMG$HOME_CURSOR (STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SMG$READ_FROM_DISPLAY (STATS_VDID,
2                               STRING)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
SIZE = 55
DO WHILE ((STRING (SIZE:SIZE) .EQ. ' ') .AND.
2         (SIZE .GT. 1))
  SIZE = SIZE - 1
END DO
```

7.4.6.2 Reading from a Virtual Keyboard

The routine SMG$CREATE_VIRTUAL_KEYBOARD establishes a device for input operations; the default device is the user's terminal. The routine SMG$READ_STRING reads characters typed on the screen until the user types a terminator or until the maximum size (which defaults to 512 characters) is exceeded. (The terminator is usually a carriage return; see the routine description in the *VMS RTL Screen Management (SMG$) Manual* for a complete list of terminators.) The current cursor location for the display determines where the read operation begins.

The VMS terminal driver processes carriage returns differently than the SMG$ routines. Therefore, in order to scroll input accurately, you must keep track of your vertical position in the display area. Explicitly set the cursor position and scroll the display. If a read operation takes place on other than the last row of the display, advance the cursor to the beginning of the next row before the next operation. If a read operation takes place on the last row of the display, scroll the display with SMG$SCROLL_DISPLAY_AREA and then set the cursor to the beginning of the row. Modify the read operation with TRM$M_TM_NOTRMECHO to ensure that no extraneous scrolling occurs.

Example 7–9 reads input until CTRL/Z is pressed:

Example 7–9 Reading Data from a Virtual Keyboard

```
    .
    .
    .
! Read first record
STATUS = SMG$HOME_CURSOR (VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (KBID,
2                         TEXT,
2                         'Prompt: ',
2                         4,
2                         TRM$M_TM_TRMNOECHO,,,
2                         TEXT_SIZE,,
2                         VDID)
```

Example 7–9 Cont'd. on next page

Example 7–9 (Cont.) Reading Data from a Virtual Keyboard

```
! Read remaining records until CTRL/Z
DO WHILE (STATUS .NE. SMG$_EOF)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Process record
   .
   .
   .
  ! Set up screen for next read
  ! Display area contains four rows
  STATUS = SMG$RETURN_CURSOR_POS (VDID, ROW, COL)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  IF (ROW .EQ. 4) THEN
    STATUS = SMG$SCROLL_DISPLAY_AREA (VDID)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = SMG$SET_CURSOR_ABS (VDID, 4, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ELSE
    STATUS = SMG$SET_CURSOR_ABS (VDID,, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = SMG$SET_CURSOR_REL (VDID, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Read next record
  STATUS = SMG$READ_STRING (KBID,
2                           TEXT,
2                           'Prompt: ',
2                           4,
2                           TRM$M_TM_TRMNOECHO,,,
2                           TEXT_SIZE,,
2                           VDID)
END DO
```

Note: Since you are controlling the scrolling, SMG$PUT_LINE and SMG$PUT_WITH_SCROLL might not scroll as expected. When scrolling a mix of input and output, you can prevent possible problems by using SMG$PUT_CHARS.

7.4.6.3 Reading from the Keypad

To read from the keypad in keypad mode (that is, pressing a keypad character to perform some special action rather than to enter data), modify the read operation with TRM$M_TM_ESCAPE and TRM$M_TM_NOECHO. Examine the terminator to determine which key was pressed.

Example 7–10 moves the cursor about on the screen in response to the user's pressing the keys surrounding the 5 key on the keypad. The 8 key moves the cursor north (up); the 9 key moves the cursor northeast; the 6 key moves the cursor east (right); and so on. The routine SMG$SET_CURSOR_REL is called, instead of invoked as a function, because you do not want to abort the program on an error. (The error attempts to move the cursor out of the display area and, if this error occurs, you do not want the cursor to move.) The read operation is also modified with TRM$M_TM_PURGE to prevent the user from getting ahead of the cursor.

Example 7–10 Reading Data from the Keypad

```
    .
    .
    .
INTEGER STATUS,
2       PBID,
2       ROWS,
2       COLUMNS,
2       VDID,      ! Virtual display ID
2       KID,       ! Keyboard ID
2       SMG$CREATE_PASTEBOARD,
2       SMG$CREATE_VIRTUAL_DISPLAY,
2       SMG$CREATE_VIRTUAL_KEYBOARD,
2       SMG$PASTE_VIRTUAL_DISPLAY,
2       SMG$HOME_CURSOR,
2       SMG$SET_CURSOR_REL,
2       SMG$READ_STRING,
2       SMG$ERASE_PASTEBOARD,
2       SMG$PUT_CHARS,
2       SMG$READ_FROM_DISPLAY
CHARACTER*31 INPUT_STRING,
2            MENU_STRING
INTEGER*2    TERMINATOR
INTEGER*4    MODIFIERS
INCLUDE '($SMGDEF)'
INCLUDE '($TRMDEF)'
! Set up screen and keyboard
STATUS = SMG$CREATE_PASTEBOARD (PBID,
2                               'SYS$OUTPUT',
2                               ROWS,
2                               COLUMNS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (ROWS,
2                                    COLUMNS,
2                                    VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PUT_CHARS (VDID,
2                       '__ MENU CHOICE ONE',
2                       10,30)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PUT_CHARS (VDID,
2                       '__ MENU CHOICE TWO',
2                       15,30)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (KID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (VDID,
2                                   PBID,
2                                   1,
2                                   1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Put cursor in NW corner
STATUS = SMG$HOME_CURSOR (VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 7–10 Cont'd. on next page

Example 7–10 (Cont.) Reading Data from the Keypad

```
! Read character from keyboard
MODIFIERS = TRM$M_TM_ESCAPE .OR.
2           TRM$M_TM_NOECHO .OR.
2           TRM$M_TM_PURGE
STATUS = SMG$READ_STRING (KID,
2                         INPUT_STRING,
2                         ,
2                         6,
2                         MODIFIERS,
2                         ,
2                         ,
2                         ,
2                         TERMINATOR)
DO WHILE ((STATUS) .AND.
2         (TERMINATOR .NE. SMG$K_TRM_CR))
  ! Check status of last read
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! North
  IF (TERMINATOR .EQ. SMG$K_TRM_KP8) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, 0)
  ! Northeast
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP9) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, 1)
  ! Northwest
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP7) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, -1)
  ! South
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP2) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, 0)
  ! Southeast
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP3) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, 1)
  ! Southwest
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP1) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, -1)
  ! East
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP6) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 0, 1)
  ! West
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP4) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 0, -1)
  END IF
  ! Read another character
  STATUS = SMG$READ_STRING (KID,
2                           INPUT_STRING,
2                           ,
2                           6,
2                           MODIFIERS,
2                           ,
2                           ,
2                           ,
2                           TERMINATOR)
END DO
```

Example 7–10 Cont'd. on next page

Example 7–10 (Cont.) Reading Data from the Keypad

```
! Read menu entry and process
!     Guidelines for reading from the display
!     are in Section 7.4.6.1.
STATUS = SMG$READ_FROM_DISPLAY (VDID,
2                                MENU_STRING)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
! Clear screen
STATUS = SMG$ERASE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

7.4.6.4 Reading Composed Input

The routine SMG$CREATE_KEY_TABLE creates a table that equates keys to character strings. When you read input using the routine SMG$READ_COMPOSED_LINE and the user presses a defined key, the corresponding character string in the table is substituted for the key. The routine SMG$ADD_KEY_DEF can be used to load the table. Composed input also permits the following:

- If states—You can define the same key to mean different things in different states. You can define a key to cause a change in state. The change in state can be temporary (until after the next defined key is pressed) or permanent (until a key that changes states is pressed).
- Input termination—You can define the key to cause termination of the input transmission (as if the RETURN key were pressed after the character string). If the key is not defined to cause termination of the input, the user must press a terminator or another key that does cause termination.

Example 7–11 defines the keys 1 through 9 on the keypad and permits the user to temporarily change state by pressing the PF1 key. Pressing the 1 key on the keypad is equivalent to typing 1000 and pressing the RETURN key. Pressing PF1 key and then 1 key on the keypad is equivalent to typing 10000 and pressing the RETURN key.

Example 7–11 Redefining Keys

```
INTEGER*4 TABLEID
    .
    .
    .
! Create table for key definitions
STATUS = SMG$CREATE_KEY_TABLE (TABLEID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Load table
! If user presses PF1, the state changes to BYTEN
! The BYTEN state is in effect only for the very next key
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'PF1',
2                         ,,,'BYTEN')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Pressing KP1 through Kp9 in the null state is like typing
! 1000 through 9000 and pressing return
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP1',
2                         ,
2                         SMG$M_KEY_TERMINATE,
2                         '1000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP2',
2                         ,
2                         SMG$M_KEY_TERMINATE,
2                         '2000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP9',
2                         ,
2                         SMG$M_KEY_TERMINATE,
2                         '9000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 7–11 Cont'd. on next page

Example 7–11 (Cont.) Redefining Keys

```
! Pressing KP1 through KP9 in the BYTEN state is like
! typing 10000 through 90000 and pressing return
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP1',
2                         'BYTEN',
2                         SMG$M_KEY_TERMINATE,
2                         '10000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP2',
2                         'BYTEN',
2                         SMG$M_KEY_TERMINATE,
2                         '20000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'KP9',
2                         'BYTEN',
2                         SMG$M_KEY_TERMINATE,
2                         '90000')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! End loading key definition table
    .
    .
    .
! Read input which substitutes key definitions where appropriate
STATUS = SMG$READ_COMPOSED_LINE (KBID,
2                                TABLEID,
2                                STRING,
2                                SIZE,
2                                VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
```

Use the routine SMG$DELETE_KEY_DEF to delete a key definition and the routine SMG$GET_KEY_DEF to examine a key definition. You can also load key definition tables with the routines SMG$DEFINE_KEY and SMG$LOAD_KEY_DEFS; the input to these routines is in the form of DCL DEFINE/KEY commands.

To use keypad keys 0 through 9, the keypad must be in application mode. (Use the /APPLICATION qualifier with the DCL command SET TERMINAL; see the *VMS DCL Dictionary* for details.)

7.4.7 Controlling Screen Updates

If your program needs to make a number of changes to a virtual display, you might want to have the SMG$ routines make all of the changes before updating the display. The routine SMG$BEGIN_DISPLAY_UPDATE causes output operations to a pasted display to be reflected only in the display's buffers. The routine SMG$END_DISPLAY_UPDATE writes the display's buffer to the pasteboard.

The SMG$BEGIN_DISPLAY_UPDATE and SMG$END_DISPLAY_UPDATE routines increment and decrement a counter. When this counter's value is 0, output to the virtual display is immediately sent to the pasteboard. The counter mechanism allows a subroutine to request and turn off batching without disturbing the batching state of the calling program.

A second set of routines, SMG$BEGIN_PASTEBOARD_UPDATE and SMG$END_PASTEBOARD_UPDATE, allow you to buffer output to a pasteboard in a similar manner.

7.4.8 Modularity

You must take care when using the SMG$ routines not to corrupt the mapping between the screen appearance and the internal representation of the screen. Therefore, observe the following guidelines:

- Mixing SMG I/O and Other Forms of I/O —In general, you should not use any other form of terminal I/O while the terminal is active as a pasteboard. If you do use I/O other than SMG I/O (for example, if you invoke a subprogram that may perform non-SMG terminal I/O), first invoke the routine SMG$SAVE_PHYSICAL_SCREEN and when the non-SMG I/O completes, invoke the routine SMG$RESTORE_PHYSICAL_SCREEN, as demonstrated in the following example:

```
STATUS = SMG$SAVE_PHYSICAL_SCREEN (PBID,
2                                  SAVE_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CALL GET_EXTRA_INFO (INFO_ARRAY)
STATUS = SMG$RESTORE_PHYSICAL_SCREEN (PBID,
2                                     SAVE_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- Sharing the pasteboard—A routine using the terminal screen without consideration for its current contents must use the existing pasteboard ID associated with the terminal (therefore, a program unit invoking a subprogram that also performs screen I/O must pass the pasteboard ID) and delete any virtual displays it creates before returning control to the higher level code. The safest way to clean up your virtual displays is to call the routine SMG$POP_VIRTUAL_DISPLAY and name the first virtual display you created. The following example invokes a subprogram that uses the terminal screen:

Invoking Program Unit

```
CALL GET_EXTRA_INFO (PBID,
2                    INFO_ARRAY)
    .
    .
    .
CALL STATUS = SMG$CREATE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Subprogram

```
SUBROUTINE GET_EXTRA_INFO (PBID,
2                          INFO_ARRAY)
    .
    .
    .
! Start executable code
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (4,
2                                    40,
2                                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (INSTR_VDID,
2                                   PBID, 1, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
STATUS = SMG$POP_VIRTUAL_DISPLAY (INSTR_VDID,
2                                 PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END
```

- Sharing virtual displays—To share a virtual display created by higher level code, the lower level code must use the virtual display ID created by the higher level code; an invoking program unit must pass the virtual display ID to the subprogram. To share a virtual display created by lower level code, the higher level code must use the virtual display ID created by the lower level code; a subprogram must return the virtual display ID to the invoking program. The following example permits a subprogram to use a virtual display created by the invoking program unit:

Invoking Program Unit

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (4,
2                                    40,
2                                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (INSTR_VDID,
2                                   PBID, 1, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CALL GET_EXTRA_INFO (PBID,
2                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Subprogram

```
SUBROUTINE GET_EXTRA_INFO (PBID,
2                          INSTR_VDID)
```

7.5 Special Input/Ouput Actions

Screen management input routines and the SYS$QIO and SYS$QIOW system services allow you to perform I/O operations otherwise unavailable to high-level languages. For example, you can allow a user to interrupt normal program execution by typing a character and by having a mechanism for reading that character. You can also control such things as echoing, time allowed for input, and whether data is read from the type-ahead buffer.

Some of the operations described in the following sections require the use of the SYS$QIO or SYS$QIOW system services. For more information on the QIO system services, see the *VMS System Services Reference Manual.*

Other operations described in the following sections can be performed by calling the SMG$ input routines. The SMG$ input routines can be used alone or with the SMG$ output routines. Section 7.4 describes how to use the input routines with the output routines. This section assumes that you are using the input routines alone. To use the SMG$ input routines, do the following:

1. Call SMG$CREATE_VIRTUAL_KEYBOARD to associate a logical keyboard with a device or file specification (SYS$INPUT by default). SMG$CREATE_VIRTUAL_KEYBOARD returns a keyboard identification number; use that number to identify the device or file to the SMG input routines.
2. Call an SMG$ input routine (SMG$READ_STRING or SMG$READ_COMPOSED_LINE) to read data typed at the device associated with the virtual keyboard.

When using the SMG$ input routines without the SMG$ output routines, do not specify the optional argument of the input routine.

7.5.1 CTRL/C and CTRL/Y Interrupts

The QIO system services enable you to detect a CTRL/C or CTRL/Y interrupt at a user terminal, even if you have not issued a read to the terminal. To do so, you must take the following steps:

1. Queue an asynchronous system trap (AST)—Issue the SYS$QIO or SYS$QIOW system service with a function code of IO$_SETMODE modified by either IO$M_CTRLCAST (for CTRL/C interrupts) or IO$M_CTRLYAST (for CTRL/Y interrupts). For the **P1** argument, provide the name of a subroutine to be executed when the interrupt occurs. For the **P2** argument, you can optionally identify one longword argument to pass to the AST subroutine.
2. Write an AST subroutine—Write the subroutine identified in the **P1** argument of the QIO system service and link the subroutine into your program. Your subroutine can take one longword dummy argument to be associated with the **P2** argument in the QIO system service. You must define common areas to access any other data in your program from the AST routine.

If you enter CTRL/C or CTRL/Y after your program queues the appropriate AST, the system interrupts your program and transfers control to your AST subroutine (this action is called delivering the AST). After your AST subroutine executes, the system returns control to your program at the point of interruption (unless your AST subroutine causes the program to exit or another AST has been queued). Note the following guidelines in using CTRL/C and CTRL/Y ASTs:

- ASTs are asynchronous—Since your AST subroutine does not know exactly where you are in your program when the interrupt occurs, you should avoid manipulating data or performing other mainline activities. In general, the AST subroutine should notify the mainline code (for example, by setting a flag) that the interrupt occurred or clean up and exit from the program (if that is what you want to do).
- ASTs need new channels to the terminal—If you try to access the terminal with language I/O statements using SYS$INPUT or SYS$OUTPUT, you may receive a redundant I/O error. You must establish another channel to the terminal by explicitly opening the terminal.
- CTRL/C and CTRL/Y ASTs are one-time ASTs—After a CTRL/C or CTRL/Y AST is delivered, it is dequeued. You must reissue the QIO system service if you wish to trap another interrupt.
- Many ASTs can be queued—You can queue multiple ASTs (for the same or different AST subroutines, on the same or different channels) by issuing the appropriate number of QIO system services. The system delivers the ASTs on a last-in first-out basis.
- Unhandled CTRL/Cs turn into CTRL/Ys—If the user enters CTRL/C and you do not have an AST queued to handle the interrupt, the system turns the CTRL/C interrupt into a CTRL/Y interrupt.
- DCL handles CTRL/Y interrupts—DCL handles CTRL/Y interrupts by returning the user to DCL command level, where the user has the option of continuing or exiting from your program. DCL takes precedence over your AST subroutine for CTRL/Y interrupts. Your CTRL/Y AST subroutine is executed only under the following circumstances: (1) if CTRL/Y interrupts are disabled at DCL level (SET NOCONTROL_Y) before your program is executed, (2) if your program disables DCL CTRL/Y interrupts with LIB$DISABLE_CTRL, or (3) if the user elects to continue your program after DCL interrupts it.
- You can dequeue CTRL/C and CTRL/Y ASTs—You can dequeue all CTRL/C or CTRL/Y ASTs on a channel by issuing the appropriate QIO system service with a value of 0 for **P1** (passed by immediate value). You can dequeue all CTRL/C ASTs on a channel by issuing the SYS$CANCEL system service for the appropriate channel. You can dequeue all CTRL/Y ASTs on a channel by issuing the SYS$DASSGN system service for the appropriate channel.
- You can use SMG$ routines—You can connect to the terminal using the SMG$ routines from either AST level or mainline code. Do not attempt to connect to the terminal from AST level if you do so in your mainline code.

Example 7–12 permits the terminal user to interrupt a display to see how many lines have been typed up to that point.

Example 7–12 Using Interrupts to Perform I/O

```
!Main Program
    .
    .
    .
INTEGER STATUS
! Accumulated data records
CHARACTER*132 STORAGE (255)
INTEGER*4     STORAGE_SIZE (255),
2             STORAGE_COUNT
! QIOW and QIO structures
INTEGER*2 INPUT_CHAN
INTEGER*4 CODE
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT
  BYTE      TRANSMIT,
2           RECEIVE,
2           CRFILL,
2           LFFILL,
2           PARITY,
2           ZERO
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Flag to notify program of CTRL/C interrupt
LOGICAL*4 CTRLC_CALLED
! AST subroutine to handle CTRL/C interrupt
EXTERNAL CTRLC_AST
! Subroutines
INTEGER SYS$ASSIGN,
2       SYS$QIOW
! Symbols used for I/O operations
INCLUDE '($IODEF)'
! Put values into array
CALL LOAD_STORAGE (STORAGE,
2                  STORAGE_SIZE,
2                  STORAGE_COUNT)
! Assign channel and set up QIOW structures
STATUS = SYS$ASSIGN ('SYS$INPUT',
2                    INPUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CODE = IO$_SETMODE .OR. IO$M_CTRLCAST
! Queue an AST to handle CTRL/C interrupt
STATUS = SYS$QIOW (,
2                  %VAL (INPUT_CHAN),
2                  %VAL (CODE),
2                  IOSB,
2                  ,,
2                  CTRLC_AST,     ! Name of AST routine
2                  CTRLC_CALLED, ! Argument for AST routine
2                  ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (.NOT. IOSB.IOSTAT)
2  CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
```

Example 7–12 Cont'd. on next page

Example 7–12 (Cont.) Using Interrupts to Perform I/O

```
! Display STORAGE array, one element per line
DO I = 1, STORAGE_COUNT
  TYPE *, STORAGE (I) (1:STORAGE_SIZE (I))

  ! Additional actions if user types CTRL/C
  IF (CTRLC_CALLED) THEN
    CTRLC_CALLED = .FALSE.
    ! Show user number of lines displayed so far
    TYPE *, 'Number of lines: ', I
    ! Requeue AST
    STATUS = SYS$QIOW (,
2                      %VAL (INPUT_CHAN),
2                      %VAL (CODE),
2                      IOSB,
2                      ,,
2                      CTRLC_AST,
2                      CTRLC_CALLED,
2                      ,,,)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    IF (.NOT. IOSB.IOSTAT)
2      CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
  END IF
END DO

END
```

AST Routine

```
! AST routine
! Notifies program that user typed CTRL/C
SUBROUTINE CTRLC_AST (CTRLC_CALLED)
LOGICAL*4 CTRLC_CALLED
CTRLC_CALLED = .TRUE.

END
```

7.5.2 Unsolicited Input

You can detect input from the terminal even if you have not called SMG$READ_COMPOSED_LINE or SMG$READ_STRING by using SMG$ENABLE_UNSOLICITED_INPUT. This routine uses the AST mechanism to transfer control to a subprogram of your choice each time the user types at the terminal; the AST subprogram is responsible for reading any input. When the subprogram completes, control returns to your mainline code where it was interrupted.

The SMG$ENABLE_UNSOLICITED_INPUT is not an SMG$ input routine. Before invoking SMG$ENABLE_UNSOLICITED_INPUT, you must invoke SMG$CREATE_PASTEBOARD to associate a pasteboard with the terminal and SMG$CREATE_VIRTUAL_KEYBOARD to associate a virtual keyboard with the same terminal.

SMG$ENABLE_UNSOLICITED_INPUT accepts the following arguments:

- The pasteboard identification number (use the value returned by SMG$CREATE_PASTEBOARD)

- The name of an AST subprogram
- An argument to be passed to the AST subprogram

When SMG$ENABLE_UNSOLICITED_INPUT invokes the AST subprogram, it passes two arguments to the subprogram: the pasteboard identification number and the argument that you specified. Typically, you write the AST subprogram to read the unsolicited input with SMG$READ_STRING. Since SMG$READ_STRING requires that you specify the virtual keyboard at which the input was typed, specify the virtual keyboard identification number as the second argument to pass to the AST subprogram.

Example 7–13 permits the terminal user to interrupt the display of a series of arrays and either go on to the next array (by typing input beginning with an uppercase N) or exit from the program (by typing input beginning with anything else).

Example 7–13 Receiving Unsolicited Input from a Virtual Keyboard

```
! Main Program
! The main program calls DISPLAY_ARRAY once for each array.
! DISPLAY_ARRAY displays the array in a DO loop.
! If the user enters input from the terminal, the loop is
! interrupted and the AST routine takes over.
! If the user types anything beginning with an N, the AST
! sets DO_NEXT and resumes execution -- DISPLAY_ARRAY drops
! out of the loop processing the array (because DO_NEXT is
! set -- and the main program calls DISPLAY_ARRAY for the
! next array.
! If the user types anything not beginning with an N,
! the program exits.
    .
    .
    .
INTEGER*4 STATUS,
2         VKID,  ! Virtual keyboard ID
2         PBID   ! Pasteboard ID
! Storage arrays
INTEGER*4 ARRAY1 (256),
2         ARRAY2 (256),
2         ARRAY3 (256)
! System routines
INTEGER*4 SMG$CREATE_PASTEBOARD,
2         SMG$CREATE_VIRTUAL_KEYBOARD,
2         SMG$ENABLE_UNSOLICITED_INPUT
! AST routine
EXTERNAL  AST_ROUTINE
```

Example 7–13 Cont'd. on next page

Example 7–13 (Cont.) Receiving Unsolicited Input from a Virtual Keyboard

```
! Create a pasteboard
STATUS = SMG$CREATE_PASTEBOARD (PBID,        ! Pasteboard ID
2                                'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create a keyboard for the same device
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,  ! Keyboard ID
2                                    'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Enable unsolicited input
STATUS = SMG$ENABLE_UNSOLICITED_INPUT (PBID, ! Pasteboard ID
2                                     AST_ROUTINE,
2                                     VKID) ! Pass keyboard
                                            ! ID to AST
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
! Call display subroutine once for each array
CALL DISPLAY_ARRAY (ARRAY1)
CALL DISPLAY_ARRAY (ARRAY2)
CALL DISPLAY_ARRAY (ARRAY3)

END
```

Array Display Routine

```
! Subroutine to display one array
SUBROUTINE DISPLAY_ARRAY (ARRAY)
! Dummy argument
INTEGER*4 ARRAY (256)
! Status
INTEGER*4 STATUS
! Flag for doing next array
LOGICAL*4 DO_NEXT
COMMON /DO_NEXT/ DO_NEXT
! If AST has been delivered, reset
IF (DO_NEXT) DO_NEXT = .FALSE.
! Initialize control variable
I = 1
! Display entire array unless interrupted by user
! If interrupted by user (DO_NEXT is set), drop out of loop
DO WHILE ((I .LE. 256) .AND. (.NOT. DO_NEXT))
  TYPE *, ARRAY (I)
  I = I + 1
END DO

END
```

Example 7–13 Cont'd. on next page

Example 7–13 (Cont.) Receiving Unsolicited Input from a Virtual Keyboard

AST Routine

```
! Subroutine to read unsolicited input
SUBROUTINE AST_ROUTINE (PBID,
2                       VKID)
! dummy arguments
INTEGER*4 PBID,                  ! Pasteboard ID
2         VKID                   ! Keyboard ID
! Status
INTEGER*4 STATUS
! Flag for doing next array
LOGICAL*4 DO_NEXT
COMMON /DO_NEXT/ DO_NEXT
! Input string
CHARACTER*4 INPUT
! Routines
INTEGER*4 SMG$READ_STRING
! Read input
STATUS = SMG$READ_STRING (VKID,  ! Keyboard ID
2                         INPUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! If user types anything beginning with N, set DO_NEXT
! otherwise, exit from program
IF (INPUT (1:1) .EQ. 'N') THEN
  DO_NEXT = .TRUE.
ELSE
  CALL EXIT
END IF

END
```

7.5.3 Type-Ahead Buffer

Normally, if the user types on the terminal before your application is able to read from that device, the input is saved in a special data structure maintained by the system called the type-ahead buffer. When your application is ready to read from the terminal, the input is transferred from the type-ahead buffer to your input buffer. The type-ahead buffer is preset at a size of 78 bytes. If the HOSTSYNC characteristic is on (the usual condition), input to the type-ahead buffer is stopped (the keyboard locks) when the buffer is within eight bytes of becoming full. If the HOSTSYNC characteristic is off, the bell rings when the type-ahead buffer is within eight bytes of becoming full; if you overflow the buffer, the excess data is lost. The system parameter TTY_ALTALARM determines the point at which input is stopped or the bell rings.

You can clear the type-ahead buffer by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_PURGE in the **modifiers** argument. Clearing the type-ahead buffer has the effect of reading only what the user types on the terminal after the read operation is invoked. Any characters in the type-ahead buffer are lost. The following example illustrates how to purge the type-ahead buffer:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,       ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,
2                                     'SYS$INPUT') ! I/O device
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,      ! Keyboard ID
2                         INPUT,     ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_PURGE,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also clear the type-ahead buffer with a QIO read operation modified by IO$M_PURGE (defined in $IODEF). You can turn off the type-ahead buffer for further read operations with a QIO set mode operation that specifies TT$M_NOTYPEAHD as a basic terminal characteristic.

You can examine the type-ahead buffer by issuing a QIO sense mode operation modified by IO$M_TYPEAHDCNT. The number of characters in the type-ahead buffer and the value of the first character are returned to the **P1** argument.

The size of the type-ahead buffer is determined by the system parameter TTY_TYPAHDSZ. You can specify an alternate type-ahead buffer by turning on the ALTYPEAHD terminal characteristic; the size of the alternate type-ahead buffer is determined by the system parameter TTY_ALTYPAHD.

7.5.4 Echo

Normally, the system writes back to the terminal any printable characters that the user types on that terminal. The system also writes highlighted words in response to certain control characters; for example, the system writes EXIT if the user enters CTRL/Z. If the user types ahead of your read, the characters are not echoed until you read them from the type-ahead buffer.

You can turn off echoing when you invoke a read operation by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_NOECHO in the **modifiers** argument. You can turn off echoing for control characters only by modifying the read operation with TRM$M_TM_TRMNOECHO. The following example turns off all echoing for the read operation:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,        ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,        ! Keyboard ID
2                                     'SYS$INPUT') ! I/O device
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,        ! Keyboard ID
2                         INPUT,       ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_NOECHO,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also turn off echoing with a QIO read operation modified by IO$M_NOECHO (defined in $IODEF). You can turn off echoing for further read operations with a QIO set mode operation that specifies TT$M_NOECHO as a basic terminal characteristic.

7.5.5 Timeout

Using SMG$READ_STRING, you can restrict the user to a certain amount of time in which to respond to a read command. If your application reads data from the terminal using SMG$READ_STRING, you can modify the timeout characteristic by specifying, in the **timeout** argument, the number of seconds the user has to respond. If the user fails to type a character in the allotted time, the error condition SS$_TIMEOUT (defined in $SSDEF) is returned. The following example restricts the user to 8 seconds in which to respond to a read command:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,      ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($SSDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,      ! Keyboard ID
2                         INPUT,     ! Data read
2                         'Prompt> ',
2                         512,
2                         ,
2                         8,
2                         ,
2                         INPUT_SIZE)
IF (.NOT. STATUS) THEN
  IF (STATUS .EQ. SS$_TIMEOUT) CALL NO_RESPONSE ()
ELSE
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

You can cause a QIO read operation to time out after a certain number of seconds by modifying the operation with IO$M_TIMED and by specifying the number of seconds as the **P3** argument. A message broadcast to a terminal resets a timer set for a timed read operation (regardless of whether the operation was initiated with QIO or SMG).

Note that the timed read operations mentioned above work on a character-by-character basis. To set a time limit on an input record rather than an input character, you must use the SYS$SETIMR system service. The SYS$SETIMR executes an AST routine at a specified time. The specified time is the input time limit. When the specified time is reached, the AST routine cancels any outstanding I/O on the channel assigned to the user's terminal.

7.5.6 Lowercase to Uppercase Conversion

You can automatically convert user input to uppercase (that is, any lowercase characters typed by the user are transformed to uppercase) by reading from the terminal with the SMG$READ_STRING routine by specifying TRM$M_TM_CVTLOW in the **modifiers** argument,

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,      ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,  ! Keyboard ID
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,     ! Keyboard ID
2                         INPUT,    ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_CVTLOW,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also convert lowercase characters with a QIO read operation modified by IO$M_CVTLOW (defined in $IODEF).

7.5.7 Line Editing and Control Actions

Normally, the user can edit input as explained in the *VAX EDT Reference Manual*. You can inhibit line editing on the read operation by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_NOFILTR in the **modifiers** argument. The following example shows how you can inhibit line editing:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,      ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,  ! Keyboard ID
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,    ! Keyboard ID
2                         INPUT,  ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_NOFILTR,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also inhibit line editing with a QIO read operation modified by IO$M_NOFILTR (defined in $IODEF).

7.5.8 Broadcasts

You can write (that is, broadcast) to any interactive terminal using the SYS$BRKTHRU system service. The following example broadcasts a message to all terminals on which users are currently logged in. Use of SYS$BRKTHRU to write to a terminal allocated to a process other than your own requires OPER privilege.

```
INTEGER*4 STATUS,
2         SYS$BRKTHRUW
INTEGER*2 B_STATUS (4)
INCLUDE   '($BRKDEF)'
STATUS = SYS$BRKTHRUW (,
2                      'Accounting system started',,
2                      %VAL (BRK$C_ALLUSERS),
2                      B_STATUS,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.5.8.1 Default Handling of Broadcasts

If the terminal user has taken no action to handle broadcasts, a broadcast is written to the terminal screen at the current position (after a carriage return and line feed). If a write operation is in progress, the broadcast occurs after the write ends. If a read operation is in progress, the broadcast occurs immediately; after the broadcast, any echoed user input to the aborted read operation is written to the screen (same effect as pressing CTRL/R).

7.5.8.2 How to Create Alternate Broadcast Handlers

You can handle broadcasts to the terminal on which your program is running with SMG$SET_BROADCAST_TRAPPING. This routine uses the AST mechanism to transfer control to a subprogram of your choice each time a broadcast message is sent to the terminal; when the subprogram completes, control returns to your main line code where it was interrupted.

SMG$SET_BROADCAST_TRAPPING is not an SMG$ input routine. Before invoking SMG$SET_BROADCAST_TRAPPING, you must invoke SMG$CREATE_PASTEBOARD to associate a pasteboard with the terminal. SMG$CREATE_PASTEBOARD returns a pasteboard identification number; pass that number to SMG$SET_BROADCAST_TRAPPING to identify the terminal in question. Read the contents of the broadcast with SMG$GET_BROADCAST_MESSAGE.

Example 7–14 demonstrates how you might trap a broadcast and write it at the bottom of the screen. For more information about the use of SMG$ pasteboards and virtual displays, see Section 7.4.

Example 7–14 Trapping Broadcast Messages

```
    .
    .
    .
INTEGER*4 STATUS,
2         PBID,                                ! Pasteboard ID
2         VDID,                                ! Virtual display ID
2         SMG$CREATE_PASTEBOARD,
2         SMG$SET_BROADCAST_TRAPPING
2         SMG$PASTE_VIRTUAL_DISPLAY
COMMON    /ID/ PBID,
2              VDID
INTEGER*2 B_STATUS (4)
INCLUDE   '($SMGDEF)'
INCLUDE   '($BRKDEF)'
EXTERNAL  BRKTHRU_ROUTINE
STATUS = SMG$CREATE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (3,        ! Height
2                                    80,       ! Width
2                                    VDID,,    ! Display ID
2                                    SMG$M_REVERSE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$SET_BROADCAST_TRAPPING (PBID,    ! Pasteboard ID
2                                       BRKTHRU_ROUTINE) ! AST
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
```

Example 7–14 Cont'd. on next page

Example 7–14 (Cont.) Trapping Broadcast Messages

```
SUBROUTINE BRKTHRU_ROUTINE ()
INTEGER*4 STATUS,
2         PBID,                                    ! Pasteboard ID
2         VDID,                                    ! Virtual display ID
2         SMG$GET_BROADCAST_MESSAGE,
2         SMG$PUT_CHARS,
2         SMG$PASTE_VIRTUAL_DISPLAY
COMMON    /ID/ PBID,
2              VDID
CHARACTER*240 MESSAGE
INTEGER*2     MESSAGE_SIZE
! Read the message
STATUS = SMG$GET_BROADCAST_MESSAGE (PBID,
2                                  MESSAGE,
2                                  MESSAGE_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Write the message to the virtual display
STATUS = SMG$PUT_CHARS (VDID,
2                       MESSAGE (1:MESSAGE_SIZE),
2                       1,                         ! Line
2                       1)                         ! Column
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Make the display visble by pasting it to the pasteboard
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (VDID,
2                                  PBID,
2                                  22,             ! Row
2                                  1)              ! Column

END
```

7.6 SYS$QIO and SYS$QIOW System Services

The QIO system services permit direct interaction with the system's terminal driver. QIOs permit some operations that cannot be performed with language I/O statements and RTL routines, reduce overhead, and permit asynchronous I/O operations. However, QIOs are device dependent.

The format for SYS$QIO is as follows:

SYS$QIO([efn],chan,func[,iosb][,astadr][,astprm] [,p1][,p2][,p3][,p4][,p5][,p6])

To read from or write to a terminal with the SYS$QIO or SYS$QIOW system service, you must first associate the terminal name with an I/O channel in the SYS$ASSIGN system service, then use the assigned channel in the SYS$QIO or SYS$QIOW system service. To read from SYS$INPUT or write to SYS$OUTPUT, specify the appropriate logical name as the terminal name in the SYS$ASSIGN system service. In general, use SYS$QIO for asynchronous operations and use SYS$QIOW for all other operations.

7.6.1 Read Operations

The SYS$QIO or SYS$QIOW system service moves one record of data from a terminal to a variable. Do not use this system service, as described here, for input from a file or nonterminal device.

For synchronous I/O (your program pauses until the I/O operation completes execution), use SYS$QIOW. For complete information about the SYS$QIO and SYS$QIOW system services, refer to the *VMS System Services Reference Manual*.

The SYS$QIOW system service places the data read in the variable passed as **P1**. The second word of the status block contains the offset from the beginning of the buffer to the terminator—hence, it equals the size of the data read. Always reference the data as a substring, using the offset to the terminator as the position of the last character (that is, the size of the substring). If you reference the entire buffer, your data will include the terminator for the operation (for example, the CR character) and any excess characters from a previous operation using the buffer. (The only exception to the substring guideline is if you deliberately overflow the buffer to terminate the I/O operation.)

Example 7–15 reads a line of data from the terminal and waits for the I/O to complete.

Example 7–15 Reading Data from the Terminal Synchronously

```
    .
    .
    .
INTEGER STATUS
! QIOW structures
INTEGER*2 INPUT_CHAN              ! I/O channel
INTEGER CODE,                     ! Type of I/O operation
2       INPUT_BUFF_SIZE,          ! Size of input buffer
2       PROMPT_SIZE,              ! Size of prompt
2       INPUT_SIZE                ! Size of input line as read
PARAMETER (PROMPT_SIZE = 13,
2          INPUT_BUFF_SIZE = 132)
CHARACTER*132 INPUT
CHARACTER*(*) PROMPT
PARAMETER (PROMPT = 'Input value: ')
! Define symbols used in I/O operations
INCLUDE '($IODEF)'
! Status block for QIOW
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,               ! Return status
2         TERM_OFFSET,            ! Location of line terminator
2         TERMINATOR,             ! Value of terminator
2         TERM_SIZE               ! Size of terminator
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
```

Example 7–15 Cont'd. on next page

Example 7–15 (Cont.) Reading Data from the Terminal Synchronously

```
! Subprograms
INTEGER*4 SYS$ASSIGN,
2         SYS$QIOW
    .
    .
    .
! Assign an I/O channel to SYS$INPUT
STATUS = SYS$ASSIGN ('SYS$INPUT',
2                    INPUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Read with prompt
CODE = IO$_READPROMPT
STATUS = SYS$QIOW (,
2                  %VAL (INPUT_CHAN),
2                  %VAL (CODE),
2                  IOSB,
2                  ,,
2                  %REF (INPUT),
2                  %VAL (INPUT_BUFF_SIZE),
2                  ,,
2                  %REF (PROMPT),
2                  %VAL (PROMPT_SIZE))
! Check QIOW status
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Check status of I/O operation
IF (.NOT. IOSB.IOSTAT) CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
! Set size of input string
INPUT_SIZE = IOSB.TERM_OFFSET
    .
    .
    .
```

To perform an asynchronous read operation, use the SYS$QIO system service and specify an event flag (the first argument, which must be passed by value). Your program continues while the I/O is taking place. When you need the input from the I/O operation, invoke the SYS$SYNCH system service to wait for the event flag and status block specified in the SYS$QIO system service. If the I/O is not complete, your program pauses until it is. In this manner, you can overlap processing within your program. Naturally, you must take care not to use data returned by the I/O operation before issuing SYS$SYNCH. Example 7–16 demonstrates an asynchronous read operation.

Example 7–16 Reading Data from the Terminal Asynchronously

```
    .
    .
    .
INTEGER STATUS
! QIO structures
INTEGER*2 INPUT_CHAN     ! I/O channel
INTEGER CODE,            ! Type of I/O operation
2       INPUT_BUFF_SIZE, ! Size of input buffer
2       PROMPT_SIZE,     ! Size of prompt
2       INPUT_SIZE       ! Size of input line as read
PARAMETER (INPUT_BUFF_SIZE = 132,
2          PROMPT = 13)
CHARACTER*132 INPUT
CHARACTER*(*) PROMPT
PARAMETER (PROMPT = 'Input value: ')
INCLUDE '($IODEF)'        ! Symbols used in I/O operations
! Status block for QIO
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,       ! Return status
2           TERM_OFFSET,  ! Location of line terminator
2           TERMINATOR,   ! Value of terminator
2           TERM_SIZE     ! Size of terminator
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Event flag for I/O
INTEGER INPUT_EF
! Subprograms
INTEGER*4 SYS$ASSIGN,
2         SYS$QIO,
2         SYS$SYNCH,
2         LIB$GET_EF
    .
    .
    .
! Assign an I/O channel to SYS$INPUT
STATUS = SYS$ASSIGN ('SYS$INPUT',
2                    INPUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get an event flag
STATUS = LIB$GET_EF (INPUT_EF)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Read with prompt
CODE = IO$_READPROMPT
STATUS = SYS$QIO (%VAL (INPUT_EF),
2                 %VAL (INPUT_CHAN),
2                 %VAL (CODE),
2                 IOSB,
2                 ,,
2                 %REF (INPUT),
2                 %VAL (INPUT_BUFF_SIZE),
2                 ,,
2                 %REF (PROMPT),
2                 %VAL (PROMPT_SIZE))
```

Example 7–16 Cont'd. on next page

Example 7–16 (Cont.) Reading Data from the Terminal Asynchronously

```
! Check status of QIO
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    .
    .
    .
STATUS = SYS$SYNCH (%VAL (INPUT_EF),
2                   IOSB)
! Check status of SYNCH
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Check status of I/O operation
IF (.NOT. IOSB.IOSTAT) CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
! Set size of input string
INPUT_SIZE = IOSB.TERM_OFFSET
    .
    .
    .
```

Be sure to check the status of the I/O operation as returned in the I/O status block. In an asynchronous operation, you can only check this status after the I/O operation is complete (that is, after the call to SYS$SYNCH).

7.6.2 Write Operations

The SYS$QIO or SYS$QIOW system service moves one record of data from a character value to the terminal. Do not use this system service, as described here, for output to a file or nonterminal device.

For synchronous I/O (your program pauses until the I/O completes), use SYS$QIOW and omit the first argument (the event flag number). For complete information about SYS$QIO and SYS$QIOW, please refer to the *VMS System Services Reference Manual.*

Example 7–17 writes a line of character data to the terminal.

Example 7–17 Writing Character Data to a Terminal

```
INTEGER STATUS,
2       ANSWER_SIZE
CHARACTER*31 ANSWER
INTEGER*2 OUT_CHAN
! Status block for QIO
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,
2           BYTE_COUNT,
2           LINES_OUTPUT
  BYTE      COLUMN,
2           LINE
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Routines
INTEGER SYS$ASSIGN,
2       SYS$QIOW
```

Example 7–17 Cont'd. on next page

Example 7–17 (Cont.) Writing Character Data to a Terminal

```
! IO$ symbol definitions
INCLUDE '($IODEF)'
    .
    .
    .
STATUS = SYS$ASSIGN ('SYS$OUTPUT',
2                    OUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SYS$QIOW (,
2                  %VAL (OUT_CHAN),
2                  %VAL (IO$_WRITEVBLK),
2                  IOSB,
2                  ,
2                  ,
2                  %REF ('Answer: '//ANSWER(1:ANSWER_SIZE)),
2                  %VAL (8+ANSWER_SIZE),
2                  ,
2                  %VAL (32),,) ! Single spacing
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (.NOT. IOSB.IOSTAT) CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
END
```

7.6.3 Checking the Device Type

You are restricted to a terminal device in a QIO operation. If the user of your program redirects SYS$INPUT or SYS$OUTPUT to a file or nonterminal device, an error occurs. You can use the SYS$GETDVIW system service to make sure the logical name is associated with a terminal, as shown in Example 7-18. SYS$GETDVIW returns a status of SS$_IVDEVNAM if the logical name is defined as a file or otherwise does not equate to a device name. The type of device is the response associated with the DVI$_DEVCLASS request code and should be DC$_TERM for a terminal.

Example 7–18 Using SYS$GETDVIW to Verify the Device Name

```
RECORD /ITMLST/ DVI_LIST
LOGICAL*4 STATUS
! GETDVI buffers
INTEGER CLASS,              ! Response buffer
2       CLASS_LEN           ! Response length
! GETDVI symbols
INCLUDE '($DCDEF)'
INCLUDE '($SSDEF)'
INCLUDE '($DVIDEF)'
! Define subprograms
INTEGER SYS$GETDVIW
```

Example 7–18 Cont'd. on next page

Example 7–18 (Cont.) Using SYS$GETDVIW to Verify the Device Name

```
! Find out the device class of SYS$INPUT
DVI_LIST.BUFLEN = 4
DVI_LIST.CODE = DVI$_DEVCLASS
DVI_LIST.BUFADR = %LOC (CLASS)
DVI_LIST.RETLENADR = %LOC (CLASS_LEN)
STATUS = SYS$GETDVIW (,,'SYS$INPUT',
2                      DVI_LIST,,,,,)
IF ((.NOT. STATUS) .AND. (STATUS .NE. SS$_IVDEVNAM)) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Make sure device is a terminal
IF ((STATUS .NE. SS$_IVDEVNAM) .AND. (CLASS .EQ. DC$_TERM)) THEN
    .
    .
    .
ELSE
  TYPE *, 'Input device not a terminal'
END IF
```

7.6.4 Terminal Characteristics

The *VMS I/O User's Reference Volume* describes device-specific characteristics associated with terminals. To examine a characteristic, issue a QIO system service with the IO$_SENSEMODE function and examine the appropriate bit in the structure returned to the **P1** argument. To change a characteristic:

1. Issue a QIO system service with the IO$_SENSEMODE function.
2. Set or clear the appropriate bit in the structure returned to the **P1** argument.
3. Issue a QIO system service with the IO$_SETMODE function passing, as the **P1** argument, the structure you obtained from the sense mode operation and modified.

Example 7–19 turns off the HOSTSYNC terminal characteristic. To check that NOHOSTSYNCH has been set, enter the SHOW TERMINAL command.

Example 7–19 Disabling the HOSTSYNCH Terminal Characteristic

```
   .
   .
   .
INTEGER*4 STATUS
! I/O channel
INTEGER*2 INPUT_CHAN
! I/O status block
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT
  BYTE      TRANSMIT,
2           RECEIVE,
2           CRFILL,
2           LFFILL,
2           PARITY,
2           ZERO
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Characteristics buffer
! Note: basic characteristics are first three
!       bytes of second longword -- length is
!       last byte
STRUCTURE /CHARACTERISTICS/
  BYTE      CLASS,
2           TYPE
  INTEGER*2 WIDTH
  UNION
   MAP
    INTEGER*4 BASIC
   END MAP
   MAP
    BYTE LENGTH(4)
   END MAP
  END UNION
  INTEGER*4 EXTENDED
END STRUCTURE
RECORD /CHARACTERISTICS/ CHARBUF
! Define symbols used for I/O and terminal operations
INCLUDE '($IODEF)'
INCLUDE '($TTDEF)'
! Subroutines
INTEGER*4 SYS$ASSIGN,
2         SYS$QIOW
```

Example 7–19 Cont'd. on next page

Example 7–19 (Cont.) Disabling the HOSTSYNCH Terminal Characteristic

```
! Assign channel to terminal
STATUS = SYS$ASSIGN ('SYS$INPUT',
2                    INPUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get current characteristics
STATUS = SYS$QIOW (,
2                  %VAL (INPUT_CHAN),
2                  %VAL (IO$_SENSEMODE),
2                  IOSB,,,
2                  CHARBUF,          ! Buffer
2                  %VAL (12),,,,)    ! Buffer size
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (.NOT. IOSB.IOSTAT) CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))
! Turn off hostsync
CHARBUF.BASIC = IBCLR (CHARBUF.BASIC, TT$V_HOSTSYNC)
! Set new characteristics
STATUS = SYS$QIOW (,
2                  %VAL (INPUT_CHAN),
2                  %VAL (IO$_SETMODE),
2                  IOSB,,,
2                  CHARBUF,
2                  %VAL (12),,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (.NOT. IOSB.IOSTAT) CALL LIB$SIGNAL (%VAL (IOSB.IOSTAT))

END
```

If you modify terminal characteristics with set mode QIO operations, you should save the characteristics buffer that you obtain on the first sense mode operation and restore those characteristics with a set mode operation before exiting. (No reset is necessary if you just use modifiers on each read operation.) To ensure that the restoration is performed if the program aborts (for example, if the user presses CTRL/Y), you should restore the user's environment in an exit handler. See Chapter 9 for a description of exit handlers.

7.6.5 Record Terminators

A QIO read operation ends when the user enters a terminator or when the input buffer fills, whichever occurs first. The standard set of terminators applies unless you specify the **P4** argument in the read QIO operation. You can examine the terminator that ended the read operation by examining the input buffer starting at the terminator offset (second word of the I/O status block). The length, in bytes, of the terminator is specified by the high-order word of the I/O status block. The third word of the I/O status block contains the value of the first character of the terminator.

Examining the terminator enables you to read escape sequences from the terminal, provided that you modify the QIO read operation with the IO$M_ESCAPE modifier (or the ESCAPE terminal characteristic is set). The first character of the terminator will be the ESC character (an ASCII value of 27). The remaining characters will contain the value of the escape sequence.

7.6.6 File Terminators

You must examine the terminator to detect end-of-file (CTRL/Z) on the terminal. No error condition is generated at the QIO level. If the user presses CTRL/Z, the terminator will be the SUB character (an ASCII value of 26).

8 File I/O

I/O statements transfer data between records in files and variables in your program. The I/O statement determines the operation to be performed; the I/O control list specifies the file, record, and format attributes; and the I/O list contains the variables to be acted upon.

Some confusion might arise between records in a file and record variables. Where this chapter refers to a record variable, the term *record variable* will be used; otherwise, *record* refers to a record in a file.

8.1 File Attributes

Before writing a program that accesses a data file, you must know the attributes of the file and the order of the data. To determine this information, see your language-specific programming manual.

File attributes (organization, record structure, and so on) determine how data is stored and accessed. Typically, the attributes are specified by keywords when you open the data file.

Ordering of the data within a file is not important mechanically. However, if you attempt to read data without knowing how it is ordered within the file, you are likely to read the wrong data; if you attempt to write data without knowing how it is ordered within the file, you are likely to corrupt existing data.

8.2 File Access Strategies

When determining the file attributes and order of your data file, consider how you plan to access that data. File access strategies fall into several categories.

8.2.1 Complete Access

If your program processes all or most of the data in the file and especially if many references are made to the data, you should read the entire file into memory. Put each record in its own variable or set of variables.

If your program is larger than the amount of memory available (including the additional memory you get by using memory allocation routines), you must declare fewer variables and process your file in pieces. To determine the size of your program, add the number of bytes in each program section, or *PSECT*. The DCL command LINK/MAP produces a listing that includes the length of each PSECT.

8.2.2 Record-by-Record Access

If your program accesses records one after another or if you cannot fit the entire file into memory, you should read one record into memory at a time.

8.2.3 Discrete Records

If your program processes only a few records at a time, you should read only the necessary records into memory.

8.2.4 Sequential and Indexed Files

Use an unformatted sequential file for speed and to conserve disk space. Use indexed files to process selected sets of records or to directly access records. Use a sequential file with fixed-length records, a relative file, or an indexed file to directly access records.

8.2.5 Protection and Access

Files are owned by the process that creates them and receive the default protection of the creating process. To create a file with ownership and protection other than the default, use the FDL attributes OWNER and PROTECTION in the file.

8.2.5.1 Read Only Access

By default, the user of your program must have write access to a file in order for your program to open that file. However, if you specify the READONLY specifier when opening the file, the user only needs read access to the file in order to open it. The READONLY specifier does not set the protection on a file. The user cannot write to a file opened with the READONLY specifier.

8.2.5.2 Shared Access

The READONLY specifier and the SHARED specifier allow multiple processes to open the same file simultaneously, provided that each process uses one of these specifiers when opening the file. The READONLY specifier allows the process read access to the file; the SHARED specifier allows other processes read and write access to the file. If a process opens the file without specifying READONLY or SHARED, no other process can open that file even by specifying READONLY or SHARED.

In the following VAX FORTRAN segment, if the read operation indicates that the record is locked, the read operation is repeated. You should not attempt to read a locked record without providing a delay (in this example, the call to ERRSNS) to allow the other process time to complete its operation and unlock the record.

```
! Status variables and values
INTEGER STATUS,
2       IOSTAT,
2       IO_OK
PARAMETER (IO_OK = 0)
INCLUDE '($FORDEF)'
! Logical unit number
INTEGER LUN /1/
! Record variables
INTEGER LEN
CHARACTER*80 RECORD
    .
    .
    .
READ (UNIT = LUN,
2     FMT = '(Q,A)'
2     IOSTAT = IOSTAT) LEN, RECORD (1:LEN)
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
   IF (STATUS .EQ. FOR$_SPERECLOC) THEN
     DO WHILE (STATUS .EQ. FOR$_SPERECLOC)
     READ (UNIT = LUN,
2          FMT = '(Q,A)'
2          IOSTAT = IOSTAT) LEN, RECORD(1:LEN)
     IF (IOSTAT .NE. IO_OK) THEN
           CALL ERRSNS (,,,,STATUS)
           IF (STATUS .NE. FOR$_SPERECLOC) THEN
                CALL LIB$SIGNAL(%VAL(STATUS))
           END IF
     END IF
     END DO
ELSE
   CALL LIB$SIGNAL (%VAL(STATUS))
   END IF
END IF
    .
    .
    .
```

Each time you access a record in a shared file, that record is automatically locked until you perform another I/O operation on the same logical unit, or until you explicitly unlock the record using the UNLOCK statement. If you plan to modify a record, you should do so before unlocking it; otherwise, you should unlock the record as soon as possible.

8.2.6 Specifying File Attributes

Large sets of attributes can be specified using the File Definition Language Utility (FDL). All of the file attributes can be specified using RMS in a user-open routine (see Section 8.8). Typically, you only need programming language file specifiers. Use FDL only when language specifiers are unavailable.

Refer to the appropriate VAX programming language reference manual on the use of language specifiers.

For complete information on how to use FDL, see the *VMS File Definition Language Facility Manual*.

8.3 Loading and Unloading a Database

To copy an entire data file from the disk to program variables and back again, either use language I/O statements to read and write the data or use the SYS$CRMPSC system service to map the data. Mapping the file is faster than reading it. However, a mapped file usually uses more storage than one read using language I/O statements. Using I/O statements, you have to store only the data that you have entered. Using SYS$CRMPSC, you have to initialize the database and store the entire structure including the parts that do not yet contain data.

8.3.1 Using SYS$CRMPSC

Mapping a file means associating each byte of the file with a byte of program storage. You access data in a mapped file by referencing the program storage; your program does not use I/O statements.

Note: Files created using VMS RMS typically contain control information. Unless you are very familiar with the structure of these files, do not attempt to map one. The best practice is to map only those files that have been created as the result of mapping.

To map a file, perform the following operations:

1. Place the program variables for the data in a common block. Page align the common block at link time by specifying an options file containing the following link option as follows:

   ```
   PSECT_ATTR = name, PAGE
   ```

   The variable *name* is the name of the common block.

   Within the common block, you should specify the data in order from most complex to least complex (high to low rank) with character data last. This naturally aligns the data, thus preventing troublesome page breaks in virtual memory.

2. Open the data file using a user-open routine. The user-open routine must open the file for user I/O (as opposed to RMS I/O) and return the channel number on which the file is opened.
3. Map the data file to the common block.
4. Process the records, using the program variables in the common block.
5. Free the memory used by the common block, forcing modified data to be written back to the disk file.

Do not initialize variables in a common block that you plan to map; the initial values will be lost when SYS$CRMPSC maps the common block.

8.3.1.1 Mapping a File

The format for SYS$CRMPSC is as follows:

SYS$CRMPSC ([inadr],[retadr],[acmode],[flags],[gsdnam],[ident],[relpag],
[chan], [pagcnt],[vbn],[prot],[pfc])

For a complete description of the SYS$CRMPSC system service, see the *VMS System Services Reference Manual.*

Starting and Ending Addresses of the Map Section

Specify the location of the first variable in the common block as the value of the first array element of the array passed by the **inadr** argument and the location of the last variable in the common block as the value of the second array element. If the first variable in the common block is an array or string, the first variable in the common block is the first element of that array or string. If the last variable in the common block is an array or string, the last variable in the common block is the last element in that array or string.

Returning the Location of the Mapped Section

SYS$CRMPSC returns the location of the first and last elements mapped in the **retadr** argument. The value returned as the starting virtual address should be the same as the starting address passed to the **inadr** argument. The value returned as the ending virtual address should be equal to or slightly more than (within 512 bytes, one block) the value of the ending virtual address passed to the **inadr** argument.

If the first element is in error, you probably forgot to page align the common block containing the mapped data.

If the second element is in error, you were probably creating a new data file and forgot to specify the size of the file in your program (see Section 8.3.1.3).

Using Private Sections

Specify SEC$M_WRT for the **flags** to indicate that the section is writable. If the file is new, also specify SEC$M_DZRO to indicate that the section should be initialized to zero.

Obtaining the Channel Number

You must use a user-open routine to get the channel number (see Section 8.3.1.2). Pass the channel number to the **chan** argument.

Example 8–1 maps a data file consisting of one longword and three real arrays to the INC_DATA common block. The options file INCOME.OPT page aligns the INC_DATA common block.

If SYS$CRMPSC returns a status of SS$_IVSECFLG and you have correctly specified the flags in the mask argument, check to see if you are passing a channel number of 0.

Example 8–1 Mapping a Data File to the Common Block

```
!INCOME.OPT

PSECT_ATTR = INC_DATA, PAGE
```

INCOME.FOR

```
! Declare variables to hold statistics
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048)
INTEGER TOTAL_HOUSES
! Declare section information
! Data area
COMMON /INC_DATA/ PERSONS_HOUSE,
2                 ADULTS_HOUSE,
2                 INCOME_HOUSE,
2                 TOTAL_HOUSES
! Addresses
INTEGER ADDR(2),
2       RET_ADDR(2)
! Section length
INTEGER SEC_LEN
! Channel
INTEGER*2 CHAN,
2         GARBAGE
COMMON /CHANNEL/ CHAN,
2                GARBAGE
! Mask values
INTEGER MASK
INCLUDE '($SECDEF)'
! User-open routines
INTEGER UFO_OPEN,
2       UFO_CREATE
EXTERNAL UFO_OPEN,
2        UFO_CREATE
! Declare logical unit number
INTEGER STATS_LUN
! Declare status variables and values
INTEGER STATUS,
2       IOSTAT,
2       IO_OK
PARAMETER (IO_OK = 0)
INCLUDE '($FORDEF)'
EXTERNAL INCOME_BADMAP
! Declare logical for INQUIRE statement
LOGICAL EXIST
! Declare subprograms invoked as functions
INTEGER LIB$GET_LUN,
2       SYS$CRMPSC,
2       SYS$DELTVA,
2       SYS$DASSGN
! Get logical unit number for STATS.SAV
STATUS = LIB$GET_LUN (STATS_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
INQUIRE (FILE = 'STATS.SAV',
2        EXIST = EXIST)
```

Example 8–1 Cont'd. on next page

Example 8–1 (Cont.) Mapping a Data File to the Common Block

```
IF (EXIST) THEN
  ! Open STATS.SAV file
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='OLD',
2       USEROPEN = UFO_OPEN)
  MASK = SEC$M_WRT
ELSE
  ! If STATS.SAV does not exist, create new database
  MASK = SEC$M_WRT .OR. SEC$M_DZRO
  SEC_LEN =
!  (address of last - address of first + size of last + 511)/512
2  ( (%LOC(TOTAL_HOUSES) - %LOC(PERSONS_HOUSE(1)) + 4 + 511)/512 )
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='NEW',
2       INITIALSIZE = SEC_LEN,
2       USEROPEN = UFO_CREATE)
END IF
! Free logical unit number and map section
CLOSE (STATS_LUN)
! ********
! MAP DATA
! ********
! Specify first and last address of section
ADDR(1) = %LOC(PERSONS_HOUSE(1))
ADDR(2) = %LOC(TOTAL_HOUSES)
! Map the section
STATUS = SYS$CRMPSC (ADDR,
2                    RET_ADDR,
2                    ,
2                    %VAL(MASK),
2                    ,,,
2                    %VAL(CHAN),
2                    ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Check for correct mapping
IF ((ADDR(1) .NE. RET_ADDR(1)) .OR.
2   (ADDR(2) .GT. RET_ADDR(2)))
2  CALL LIB$SIGNAL (%VAL (%LOC(INCOME_BADMAP)))
    .
    .
    .
                     ! Reference data using the
                     ! data structures listed
                     ! in the common block
    .
    .
    .
! Close and update STATS.SAV
STATUS = SYS$DELTVA (RET_ADDR,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$DASSGN (%VAL(CHAN))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

END
```

8.3.1.2 User-Open Routine

When you open a file for mapping, you must specify a user-open routine (Section 8.8 discusses user-open routines) to perform the following operations:

1. Set the user-file open bit (FAB$V_UFO) in the FAB options mask.
2. Open the file using SYS$OPEN for an existing file or SYS$CREATE for a new file. (Do not invoke SYS$CONNECT if you have set the user-file open bit.)
3. Return the channel number to the program unit that started the OPEN operation. The channel number is in the additional status longword of the FAB (FAB$L_STV) and must be returned in a common block.
4. Return the status of the open operation (SYS$OPEN or SYS$CREATE) as the value of the user-open routine.

After setting the user-file open bit in the FAB options mask, you cannot use language I/O statements to access data in that file. Therefore, you should free the logical unit number associated with the file. The file is still open. You access the file with the channel number.

Example 8–2 shows a user-open routine invoked by the example program in Section 8.3.1.1 if the file STATS.SAV exists. (If STATS.SAV does not exist, the user-open routine must invoke SYS$CREATE rather than SYS$OPEN.)

Example 8–2 Using a User-Open Routine

```
!UFO_OPEN.FOR

INTEGER FUNCTION UFO_OPEN (FAB,
2                          RAB,
2                          LUN)

! Include RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN
! Declare channel
INTEGER*4 CHAN
COMMON /CHANNEL/ CHAN
! Declare status variable
INTEGER STATUS
! Declare system procedures
INTEGER SYS$OPEN
! Set useropen bit in the FAB options longword
FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_UFO
```

Example 8–2 Cont'd. on next page

Example 8–2 (Cont.) Using a User-Open Routine

```
! Open file
STATUS = SYS$OPEN (FAB)
! Read channel from FAB status word
CHAN = FAB.FAB$L_STV

! Return status of open operation
UFO_OPEN = STATUS

END
```

8.3.1.3 Initializing a Mapped Database

The first time you map a file you must perform the following operations in addition to those listed at the beginning of Section 8.3.1:

- Specify the size of the file—SYS$CRMPSC maps data based on the size of the file. Therefore, when creating a file that is to be mapped, you must specify in your program a file large enough to contain all of the expected data. Figure the size of your database as follows:
  1. Find the size of the common block (in bytes)—Subtract the location of the first variable in the common block from the location of the last variable in the common block and then add the size of the last element.
  2. Find the number of blocks in the common block—Add 511 to the size and divide the result by 512 (512 bytes = 1 block).
- Initialize the file when you map it—The blocks allocated to a file might not be initialized and therefore contain random data. When you first map the file, you should initialize the mapped area to zeros by setting the SEC$V_DZRO bit in the mask argument of SYS$CRMPSC.

The user-open routine for creating a file is the same as the user-open routine for opening a file except that SYS$OPEN is replaced by SYS$CREATE.

8.3.1.4 Saving a Mapped File

To close a data file opened for user I/O, you must deassign the I/O channel assigned to that file. Before you can deassign a channel assigned to a mapped file, you must delete the virtual memory associated with the file (the memory used by the common block). When you delete the virtual memory used by a mapped file, any changes made while the file was mapped are written back to the disk file. Use the SYS$DELTVA system service to delete the virtual memory used by a mapped file. Use the SYS$DASSGN system service to deassign the I/O channel assigned to a file.

The program segment shown in Example 8–3 closes a mapped file, automatically writing any modifications back to the disk. To ensure that the proper locations are deleted, pass SYS$DELTVA the addresses returned to your program by SYS$CRMPSC rather than the addresses you passed to SYS$CRMPSC. If you want to save modifications made to the mapped section without closing the file, use the SYS$UPDSEC system service. To ensure that the proper locations are updated, pass SYS$UPDSEC the addresses returned to your program by SYS$CRMPSC rather than the addresses you passed to SYS$CRMPSC. Typically, you want to wait until the update operation completes before continuing program execution. Therefore, use the **efn** argument of SYS$UPDSEC to specify an event flag to be set when the update is complete, and wait for the system service to complete before continuing. For a complete description of the SYS$DELTVA, SYS$DASSGN,

and SYS$UPDSEC system services, see the *VMS System Services Reference Manual*.

Example 8–3 Closing a Mapped File

```
! Section address
INTEGER*4 ADDR(2),
2         RET_ADDR(2)
! Event flag
INTEGER*4 FLAG
! Status block
STRUCTURE /IO_BLOCK/
  INTEGER*2 IOSTAT,
2           HARDWARE
  INTEGER*4 BAD_PAGE
END STRUCTURE
RECORD /IO_BLOCK/ IOSTATUS
    .
    .
    .
! Get an event flag
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Update the section
STATUS = SYS$UPDSEC (RET_ADDR,
2                    ,,,
2                    %VAL(FLAG)
2                    ,
2                    IOSTATUS,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Wait for section to be updated
STATUS = SYS$SYNCH (%VAL(FLAG),
2                  IOSTATUS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
    .
    .
    .
```

8.3.1.5 Example of Per-Record Processing of Entire Database

This section provides an example, written in VAX FORTRAN, of how to open and update a sequential file. A sequential file consists of records arranged one after the other in the order in which they are written to the file. Records can only be added to the end of the file. Typically, sequential files are accessed sequentially.

Creating a Sequential File

To create a sequential file, use the OPEN statement and specify the following keywords and keyword values:

- STATUS ='NEW',
- ACCESS = 'SEQUENTIAL'
- ORGANIZATION = 'SEQUENTIAL'.

The file structure keyword ORGANIZATION also accepts the values 'INDEXED' or 'RELATIVE'.

Example 8–4 creates a sequential file of fixed-length records.

Example 8–4 Creating a Sequential File of Fixed-Length Records

```
    .
    .
    .
INTEGER STATUS,
2       LUN,
2       LIB$GET_INPUT,
2       LIB$GET_LUN,
2       STR$UPCASE
INTEGER*2     FN_SIZE,
2             REC_SIZE
CHARACTER*256 FILENAME
CHARACTER*80  RECORD
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get free unit number
STATUS = LIB$GET_LUN (LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the file
OPEN (UNIT = LUN,
2     FILE = FILENAME (1:FN_SIZE),
2     ORGANIZATION = 'SEQUENTIAL',
2     ACCESS = 'SEQUENTIAL',
2     RECORDTYPE = 'FIXED',
2     FORM = 'UNFORMATTED',
2     RECL = 20,
2     STATUS = 'NEW')
! Get the record input
STATUS = LIB$GET_INPUT (RECORD,
2                       'Input: ',
2                       REC_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
DO WHILE (REC_SIZE .NE. 0)

  ! Convert to uppercase
  STATUS = STR$UPCASE (RECORD,RECORD)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  WRITE  (UNIT=LUN) RECORD(1:REC_SIZE)
  ! Get more record input
  STATUS = LIB$GET_INPUT (RECORD,
2                         'Input: ',
2                         REC_SIZE)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END DO

END
```

Updating a Sequential File

To update a sequential file, read each record from the file, update it, and write it to a new sequential file. Updated records cannot be written back as replacement records for the same sequential file from which they were read.

Example 8–5 updates a sequential file, giving the user the option of modifying a record before writing it to the new file. The same file name is used for both files; since the new update file was opened after the old file, it has a higher version number.

Example 8–5 Updating a Sequential File

```
    .
    .
    .
INTEGER STATUS,
2       LUN1,
2       LUN2,
2       IOSTAT
INTEGER*2  FN_SIZE
CHARACTER*256 FILENAME
CHARACTER*80 RECORD
CHARACTER*80 NEW_RECORD
INCLUDE '($FORDEF)'
INTEGER*4 LIB$GET_INPUT,
2         LIB$GET_LUN,
2         STR$UPCASE
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get free unit number
STATUS = LIB$GET_LUN (LUN1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the old file
OPEN (UNIT=LUN1,
2     FILE=FILENAME (1:FN_SIZE),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=20,
2     STATUS='OLD')
! Get free unit number
STATUS = LIB$GET_LUN (LUN2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the new file
OPEN (UNIT=LUN2,
2     FILE=FILENAME (1:FN_SIZE),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=20,
2     STATUS='NEW')
```

Example 8–5 Cont'd. on next page

Example 8–5 (Cont.) Updating a Sequential File

```
! Read a record from the old file
READ (UNIT=LUN1,
2     IOSTAT=IOSTAT) RECORD
IF (IOSTAT .NE. IOSTAT_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
    CALL LIB$SIGNAL (%VAL(STATUS))
  END IF
END IF

DO WHILE (STATUS .NE. FOR$_ENDDURREA)

  TYPE *, RECORD

  ! Get record update
  STATUS = LIB$GET_INPUT (NEW_RECORD,
2                         'Update: ')
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Convert to uppercase
  STATUS = STR$UPCASE (NEW_RECORD,
2                      NEW_RECORD)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  ! Write unchanged record or updated record
  IF (NEW_RECORD .EQ. ' ' ) THEN
    WRITE (UNIT=LUN2) RECORD
  ELSE
    WRITE (UNIT=LUN2) NEW_RECORD
  END IF

  ! Read the next record
  READ (UNIT=LUN1,
2       IOSTAT=IOSTAT) RECORD
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL(STATUS))
    END IF
  END IF
END DO

END
```

8.4 Sorting and Merging Sequential Files

The Sort/Merge Utility (SORT) permits you to sort and merge records in sequential files based on one or more key fields that you specify. You can sort from one to ten input files, generating a single reordered output file. You can also merge up to ten sorted input files into a single output file.

Invoking SORT from the DCL Command Level

Use the SORT and MERGE commands to sort and merge files at the DCL command level; for details, see the description of the Sort Utility in the *VMS Sort/Merge Utility Manual.*

Invoking SORT from a Program

Use the SORT/MERGE utility (SOR) routines to sort and merge files within a program. The *VMS Utility Routines Manual* contains complete specifications for the procedures and their arguments.

8.4.1 Using the File and Record Interface

Sequential files can be sorted and merged using either a file interface or a record interface. Using the file interface, your program passes entire files to SORT and receives an entire reordered file upon completion. Using the record interface, your program passes a file to SORT one record at a time, and receives the reordered file one record at a time. Typically, the record interface is used to process individual records before or after a sort operation. These interfaces can be combined, allowing your program to pass entire files to SORT and receive individual records or vice versa.

8.4.2 Multiple Sort Operations

A program can perform multiple sort operations concurrently by specifying the context argument when calling the various SORT/MERGE utility routines. The context argument is a longword that you pass and SORT updates to keep track of concurrent sort operations. A call to SOR$END_SORT reinitializes the context argument.

8.4.3 Passing Key Information

To perform sort or merge operations, you must pass key information (**key_buffer** argument) to either the SOR$BEGIN_SORT or SOR$BEGIN_MERGE routine. The **key_buffer** argument is passed as an array of words. The first word of the array contains the number of keys to be used in the sort or merge. Each block of four words that follows describes one key (multiple keys are listed in order of their priority).

- The first word of each block describes the key datatype.
- The second word determines the sort or merge order (0 for ascending, 1 for descending).
- The third word describes the relative offset of the key (beginning at position 0).
- The fourth word describes the length of the key in bytes.

To sort or merge sequential files, you must call a specific sequence of SORT/MERGE utility routines. The routines and calling sequence depend on whether you are sorting or merging and on which interface you use.

8.4.4 Sorting with the File Interface

Perform the following steps to sort sequential files using the file interface:

1. Call SOR$PASS_FILES to pass the file specifications of the input and output files to SORT. Up to ten input files are permitted. For multiple input files, you must call SOR$PASS_FILES once for each input file. The output file must be specified in the first call. A number of optional arguments control the characteristics of the output file (see the *VMS Utility Routines Manual*).
2. Call SOR$BEGIN_SORT to pass key information. You can also specify a number of sort options, including a user-written key comparison routine (see the *VMS Utility Routines Manual*). When you are using the file interface, SOR$BEGIN_SORT opens the input and output files.
3. Call SOR$SORT_MERGE to execute the sort and direct the sorted records to the output file.
4. Call SOR$END_SORT to end the sort and close the input and output files.

Example 8–6 sorts a sequential file using the file interface.

Example 8–6 Sorting a Sequential File Using the File Interface

```
    .
    .
    .
INTEGER STATUS,
2       FN_SIZE_IN,
2       FN_SIZE_OUT,
2       LUN_IN,
2       LUN_OUT
CHARACTER*256 FILENAME_IN,
2             FILENAME_OUT
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN,
2       SOR$PASS_FILES,
2       SOR$BEGIN_SORT,
2       SOR$SORT_MERGE,
2       SOR$END_SORT
EXTERNAL DSC$K_DTYPE_T  ! Character data type definition
! Define a record
STRUCTURE /EMPLOYEE/
 CHARACTER*20 NAME       ! 1:20
 CHARACTER*20 ADDRESS    ! 21:40
 CHARACTER*19 CITY       ! 41:59
 CHARACTER*2  STATE      ! 60:61
 CHARACTER*9  ZIP_CODE   ! 62:70
END STRUCTURE
RECORD /EMPLOYEE/ TEMP
! Sort key information --- 1 key
INTEGER*2  KEY_BUFFER (5)
KEY_BUFFER (1) = 1                    ! Number of keys
KEY_BUFFER (2) = %LOC(DSC$K_DTYPE_T)  ! Character data
KEY_BUFFER (3) = 0                    ! Ascending sort
KEY_BUFFER (4) = 0                  ! Start at offset 0 (pos. 1)
KEY_BUFFER (5) = 20                 ! Length of the key
```

Example 8–6 Cont'd. on next page

Example 8–6 (Cont.) Sorting a Sequential File Using the File Interface

```
! Get input file name
STATUS = LIB$GET_INPUT (FILENAME_IN,
2                       'Input file name: ',
2                       FN_SIZE_IN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get output file name
STATUS = LIB$GET_INPUT (FILENAME_OUT,
2                       'Output file name: ',
2                       FN_SIZE_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Pass files to SORT
STATUS = SOR$PASS_FILES (FILENAME_IN (1:FN_SIZE_IN),
2                        FILENAME_OUT (1:FN_SIZE_OUT))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Pass key information to SORT
STATUS = SOR$BEGIN_SORT (KEY_BUFFER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Perform sort
STATUS = SOR$SORT_MERGE ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! End sort
STATUS = SOR$END_SORT ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

8.4.5 Sorting with the Record Interface

Perform the following steps to sort files using the record interface:

1. Call SOR$BEGIN_SORT to pass key information, the longest record length, and sort options. (Record length, the **lrl** argument, must be specified for the record interface.)
2. Call SOR$RELEASE_REC once for each record that you want to release to SORT. (Record buffer, the **desc** argument, must be specified.)
3. Call SOR$SORT_MERGE to execute the sort.
4. Call SOR$RETURN_REC once for each record that is to be returned from SORT. (Record buffer, the **desc** argument, must be specified.)
5. Call SOR$END_SORT to end the sort and close the input and output files.

Example 8–7 sorts a sequential file using the record interface. Since the SOR$RELEASE_REC and SOR$RETURN_REC routines require that you pass the record as a character string, the structure block that defines the record variable uses a union block to indicate that the record variable can be interpreted as various fields or as a single character string field.

Example 8–7 Sorting a Sequential File Using the Record Interface

```
   .
   .
   .
INTEGER STATUS,
2       FN_SIZE_IN,
2       FN_SIZE_OUT,
2       LUN_IN,
2       LUN_OUT,
2       IOSTAT
INTEGER*2 LRL/72/
CHARACTER*256 FILENAME_IN,
2             FILENAME_OUT
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN,
2       SOR$BEGIN_SORT,
2       SOR$RELEASE_REC,
2       SOR$SORT_MERGE,
2       SOR$RETURN_REC,
2       SOR$END_SORT
INCLUDE '($FORDEF)'
INCLUDE '($SSDEF)'
EXTERNAL DSC$K_DTYPE_T
PARAMETER STATUS_OK = 1
! Define a record
STRUCTURE /EMPLOYEE/
 UNION
  MAP
   CHARACTER*22 NAME                      ! 1:20
   CHARACTER*20 ADDRESS                   ! 21:40
   CHARACTER*19 CITY                      ! 41:59
   CHARACTER*2  STATE                     ! 60:61
   CHARACTER*9  ZIP_CODE                  ! 62:70
  END MAP
  MAP
   CHARACTER*72 STRING
  END MAP
 END UNION
END STRUCTURE
RECORD /EMPLOYEE/ TEMP
! Sort key information --- 1 key
INTEGER*2 KEY_BUFFER (5)
KEY_BUFFER (1) = 1                     ! Number of keys
KEY_BUFFER (2) = %LOC(DSC$K_DTYPE_T)  ! Character data
KEY_BUFFER (3) = 0                     ! Ascending sort
KEY_BUFFER (4) = 0                     ! Start at offset 0 (pos. 1)
KEY_BUFFER (5) = 22                    ! Length of the key
! Get input file name
STATUS = LIB$GET_INPUT (FILENAME_IN,
2                       'Input file name: ',
2                       FN_SIZE_IN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get output file name
STATUS = LIB$GET_INPUT (FILENAME_OUT,
2                       'Output file name: ',
2                       FN_SIZE_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–7 Cont'd. on next page

Example 8–7 (Cont.) Sorting a Sequential File Using the Record Interface

```
! Get free logical unit number
STATUS = LIB$GET_LUN (LUN_IN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the input file
OPEN (UNIT=LUN_IN,
2     FILE=FILENAME_IN (1:FN_SIZE_IN),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     STATUS='OLD')
! Get free logical unit number
STATUS = LIB$GET_LUN (LUN_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the output file
OPEN (UNIT=LUN_OUT,
2     FILE=FILENAME_OUT (1:FN_SIZE_OUT),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     STATUS='NEW')
! Give SORT key information
STATUS = SOR$BEGIN_SORT (KEY_BUFFER,
2                        LRL)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Read first record from input file
READ (UNIT=LUN_IN,
2     IOSTAT=IOSTAT) TEMP
IF (IOSTAT .NE. IOSTAT_OK) THEN
  CALL ERRSNS(,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END IF
! Pass each record to SORT
DO WHILE (STATUS .NE. FOR$_ENDDURREA)

  ! Pass the record
  STATUS = SOR$RELEASE_REC (TEMP.STRING)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  ! Read next record
  READ (UNIT=LUN_IN,
2       IOSTAT=IOSTAT) TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS(,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
  END IF
END DO
! Start sorting
STATUS = SOR$SORT_MERGE ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–7 Cont'd. on next page

Example 8–7 (Cont.) Sorting a Sequential File Using the Record Interface

```
! Release records from SORT
STATUS = SOR$RETURN_REC (TEMP.STRING)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Write records to output
DO WHILE (STATUS .NE. SS$_ENDOFFILE)

  ! Write the record to output file
  WRITE (UNIT=LUN_OUT,
2        IOSTAT=IOSTAT) TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS(,,,,STATUS)
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Release the next record
  STATUS = SOR$RETURN_REC (TEMP.STRING)
  IF ((STATUS .NE. STATUS_OK) .AND.
2     (STATUS .NE. SS$_ENDOFFILE))  THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END DO

! End SORT
STATUS = SOR$END_SORT ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

8.4.6 Merging with the File Interface

Perform the following steps to merge records using the file interface:

1. Call SOR$PASS_FILES to pass the file specifications of the input and output files to SORT. Up to ten input files are permitted. For multiple input files, you must call SOR$PASS_FILES once for each input file. The output file must be specified in the first call. A number of optional arguments control the characteristics of the output file (see the *VMS Utility Routines Manual*).
2. Call SOR$BEGIN_MERGE to pass key information and merge options. You can also specify a number of merge options, including a user-written key comparison routine (see the *VMS Utility Routines Manual*). When using the file interface, SOR$BEGIN_MERGE opens the input and output files and initializes the merge operation.
3. Call SOR$END_SORT to end the merge and close the input and output files.

Example 8–8 merges two sequential files using the file interface.

Example 8–8 Merging Sequential Files Using the File Interface

```
    .
    .
    .
INTEGER STATUS,
2       FN_SIZE_IN1,
2       FN_SIZE_IN2,
2       FN_SIZE_OUT,
2       LUN_OUT
CHARACTER*256 FILENAME_IN1,
2             FILENAME_IN2,
2             FILENAME_OUT
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN,
2       SOR$PASS_FILES,
2       SOR$BEGIN_MERGE,
2       SOR$END_SORT
EXTERNAL DSC$K_DTYPE_T
! Define a record
STRUCTURE /EMPLOYEE/
 CHARACTER*22 NAME                    ! 1:20
 CHARACTER*20 ADDRESS                 ! 21:40
 CHARACTER*19 CITY                    ! 41:59
 CHARACTER*2  STATE                   ! 60:61
 CHARACTER*9  ZIP_CODE                ! 62:70
END STRUCTURE
RECORD /EMPLOYEE/ TEMP
! SORT key information --- 1 key
INTEGER*2 KEY_BUFFER (5)
KEY_BUFFER (1) = 1                   ! Number of keys
KEY_BUFFER (2) = %LOC(DSC$K_DTYPE_T) ! Character data
KEY_BUFFER (3) = 0                   ! Ascending sort
KEY_BUFFER (4) = 0                   ! Start at offset 0 (pos. 1)
KEY_BUFFER (5) = 22                  ! Length of the key
! Get first input file name
STATUS = LIB$GET_INPUT (FILENAME_IN1,
2                       'Input file name: ',
2                       FN_SIZE_IN1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get second input file name
STATUS = LIB$GET_INPUT (FILENAME_IN2,
2                       'Input file name: ',
2                       FN_SIZE_IN2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get output file name
STATUS = LIB$GET_INPUT (FILENAME_OUT,
2                       'Output file name: ',
2                       FN_SIZE_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Pass files to SORT - output file in first call
STATUS = SOR$PASS_FILES (FILENAME_IN1 (1:FN_SIZE_IN1),
2                        FILENAME_OUT (1:FN_SIZE_OUT))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Pass second input file to SORT
STATUS = SOR$PASS_FILES (FILENAME_IN2 (1:FN_SIZE_IN2))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–8 Cont'd. on next page

Example 8–8 (Cont.) Merging Sequential Files Using the File Interface

```
! Give SORT key information
STATUS = SOR$BEGIN_MERGE (KEY_BUFFER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! End merge
STATUS = SOR$END_SORT ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

8.4.7 Merging with the Record Interface

Perform the following steps to merge sequential files using the record interface:

1 Call SOR$BEGIN_MERGE to pass the key information, the longest record length, the merge order (number of input files to be merged), and a user-written input routine. The last three arguments mentioned (**lrl**, **merge_order**, and **user_input**) are required when using the record interface; the first argument (**key_buffer**) can be omitted if you specify a key comparison routine. The user input routine must determine which input file to read, read a record, and determine the record's length.

2 Call SOR$RETURN_REC once for each record to be returned from SORT. (Record buffer, the **desc** argument, must be specified.) SOR$RETURN_REC calls the user input routine until all records are passed. When using the record interface, releasing, merging, and reading of records all occur during a call to SOR$RETURN_REC.

3 Call SOR$END_SORT to end the merge and close the input and output files.

The user input routine must accept four arguments: (1) a record buffer, (2) a file order argument (an integer passed by SORT determining which input file should be read), (3) a record length buffer (an integer), and (4) a context argument (a longword used to keep track of concurrent operations). The routine must return a status value: either SS$_NORMAL for a successful read or SS$_ENDOFFILE for an end-of-file error. SOR$BEGIN_MERGE passes any other error back to the program unit performing the merge.

Example 8–9 merges two sequential files using the record interface. Note that the common block UNIT_NUMBERS is used to pass the logical unit numbers of the input files to the input routine GET_RECORD. Since the SOR$RETURN_REC routines require that you pass the record as a character string, the structure block that defines the record variable uses a union block to indicate that the record variable can be interpreted as various fields or as a single character string field.

Example 8–9 Merging Sequential Files Using the Record Interface

```
    .
    .
    .
INTEGER STATUS,
2       GET_RECORD,
2       FN_SIZE_IN1,
2       FN_SIZE_IN2,
2       FN_SIZE_OUT,
2       STATUS_OK,
2       IOSTAT_OK,
2       LUN_IN1,
2       LUN_IN2,
2       LUN_OUT,
2       RECORD_LEN,
2       IOSTAT
PARAMETER (STATUS_OK = 1)
PARAMETER (IOSTAT_OK = 0)
INTEGER*2  LRL /72/
EXTERNAL DSC$K_DTYPE_T
LOGICAL*1 ORDER           ! Order of merge
DATA  ORDER/2/
! Common block to pass luns to subroutine
COMMON /UNIT_NUMBERS/ LUN_IN1,
2                     LUN_IN2
CHARACTER*256 FILENAME_IN1,
2             FILENAME_IN2,
2             FILENAME_OUT
EXTERNAL GET_RECORD
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN,
2       SOR$BEGIN_MERGE,
2       SOR$RETURN_REC,
2       SOR$PASS_FILES,
2       SOR$END_SORT
INCLUDE '($FORDEF)'
INCLUDE '($SSDEF)'
! Define a record
STRUCTURE /EMPLOYEE/
 UNION
  MAP
   CHARACTER*22 NAME                    ! 1:20
   CHARACTER*20 ADDRESS                 ! 21:40
   CHARACTER*19 CITY                    ! 41:59
   CHARACTER*2  STATE                   ! 60:61
   CHARACTER*9  ZIP_CODE                ! 62:70
  END MAP
  MAP
   CHARACTER*72 STRING                  ! Whole record
  END MAP
 END UNION
END STRUCTURE
```

Example 8–9 Cont'd. on next page

Example 8–9 (Cont.) Merging Sequential Files Using the Record Interface

```
RECORD /EMPLOYEE/ TEMP
! Sort key information --- 1 key
INTEGER*2  KEY_BUFFER (5)
KEY_BUFFER (1) = 1                  ! Number of keys
KEY_BUFFER (2) = %LOC(DSC$K_DTYPE_T) ! Character data
KEY_BUFFER (3) = 0                  ! Ascending sort
KEY_BUFFER (4) = 0                  ! Start at offset 0 (pos. 1)
KEY_BUFFER (5) = 22                 ! Length of the key
! Get first input file name
STATUS = LIB$GET_INPUT (FILENAME_IN1,
2                        'Input file name: ',
2                        FN_SIZE_IN1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get second input file name
STATUS = LIB$GET_INPUT (FILENAME_IN2,
2                        'Input file name: ',
2                        FN_SIZE_IN2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get output file name
STATUS = LIB$GET_INPUT (FILENAME_OUT,
2                        'Output file name: ',
2                        FN_SIZE_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get free logical unit number
STATUS = LIB$GET_LUN (LUN_IN1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the input file
OPEN (UNIT=LUN_IN1,
2     FILE=FILENAME_IN1 (1:FN_SIZE_IN1),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     STATUS='OLD')
! Get free logical unit number
STATUS = LIB$GET_LUN (LUN_IN2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the second input file
OPEN (UNIT=LUN_IN2,
2     FILE=FILENAME_IN2 (1:FN_SIZE_IN2),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     STATUS='OLD')
! Get free logical unit number
STATUS = LIB$GET_LUN (LUN_OUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–9 Cont'd. on next page

Example 8–9 (Cont.) Merging Sequential Files Using the Record Interface

```
! Open the output file
OPEN (UNIT=LUN_OUT,
2     FILE=FILENAME_OUT (1:FN_SIZE_OUT),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     STATUS='NEW')
! Begin the MERGE
STATUS = SOR$BEGIN_MERGE (KEY_BUFFER,
2                         LRL,,
2                         ORDER,,,
2                         GET_RECORD)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get, merge and release records
! RETURN_REC calls GET_RECORD for input
DO WHILE (STATUS .NE. SS$_ENDOFFILE)
  STATUS = SOR$RETURN_REC (TEMP.STRING,
2                          RECORD_LEN)
  IF (.NOT. STATUS)  THEN
    IF (STATUS .NE. SS$_ENDOFFILE)
2      CALL LIB$SIGNAL (%VAL (STATUS))
  ELSE
    ! Write the record to output file
    WRITE (UNIT=LUN_OUT,
2           IOSTAT=IOSTAT) TEMP
    IF (IOSTAT .NE. IOSTAT_OK) THEN
      CALL ERRSNS(,,,,STATUS)
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
  END IF
END DO

! End the merge
STATUS = SOR$END_SORT ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

GET_RECORD.FOR

```
INTEGER FUNCTION GET_RECORD (RECORDX,
2                            FILE_ORDER,
2                            SIZE)
INTEGER  STATUS,
2        IOSTAT,
2        STATUS_OK,
2        IOSTAT_OK,
2        MAX_NUM_FILES,
2        LUNX,
2        FILE_ORDER,
2        SIZE
PARAMETER (STATUS_OK = 1)
PARAMETER (IOSTAT_OK = 0)
PARAMETER (MAX_NUM_FILES = 2)  ! Max number of input files
```

Example 8–9 Cont'd. on next page

Example 8–9 (Cont.) Merging Sequential Files Using the Record Interface

```
INCLUDE '($SSDEF)'
INCLUDE '($FORDEF)'
COMMON /UNIT_NUMBERS/ LUN_IN1,
2                     LUN_IN2
CHARACTER*72 RECORDX           ! Record buffer
GET_RECORD = SS$_NORMAL
! Determine which input file is being read
IF (FILE_ORDER .EQ. 1) THEN
  LUNX = LUN_IN1
ELSE IF (FILE_ORDER .EQ. 2) THEN
  LUNX = LUN_IN2
ELSE IF ((FILE_ORDER .LT. 1) .OR.
2         (FILE_ORDER .GT. MAX_NUM_FILES)) THEN
  GET_RECORD = SS$_ENDOFFILE
END IF
IF (GET_RECORD .NE. SS$_ENDOFFILE) THEN
  ! Read record from input file
  READ   (UNIT=LUNX,
2         IOSTAT=IOSTAT) RECORDX
  ! Error during read
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS(,,,,STATUS)
    IF (STATUS .EQ. FOR$_ENDDURREA) THEN
      GET_RECORD = SS$_ENDOFFILE
    ELSE
      CALL LIB$SIGNAL (%VAL(STATUS))
    END IF
  END IF

  ! Successful read
  SIZE = LEN (RECORDX)
END IF
END
```

8.5 Data Compression and Expansion

To compress data in a library, use the /DATA=REDUCE qualifier of the LIBRARY command (for details, see the description of the Librarian Utility in the *VMS Librarian Utility Manual*). Once a library is reduced, the librarian automatically compresses each record entered into the library and expands each record extracted from the library. To expand the entire library, use the /DATA=EXPAND qualifier of the LIBRARY command.

You cannot compress files (except for libraries) from DCL command level. However, the DCX routines allow you to compress and expand files from within a program. (For a complete description of the DCX routines, see the *VMS Utility Routines Manual*.) To access a compressed file, you must first expand that file. Therefore, large infrequently accessed files are good candidates for compression. You can compress small files; however, it is inefficient since you must store a data compression/expansion function with the compressed records.

8.5.1 Compression Routines

Compressing a file with the DCX routines involves the following steps (an example follows):

1. Initialize an analysis work area—Use the DCX$ANALYZE_INIT routine to initialize a work area for analyzing the records. The first (and, typically, the only) argument passed to DCX$ANALYZE_INIT is an integer variable for storing the context value. The data compression facility assigns a value to the context variable and associates the value with the created work area. Each time you want to analyze a record in that area, specify the associated context variable. You can analyze two or more files at once by creating a different work area for each file, giving each area a different context variable, and analyzing the records of each file in the appropriate work area.

2. Analyze the records in the file—Use the DCX$ANALYZE_DATA routine to pass each record in the file to an analysis work area. During analysis, the data compression facility gathers information that DCX$MAKE_MAP uses to create the compression/expansion function for the file. To ensure that the first byte of each record is passed to the data compression facility rather than being interpreted as a carriage control, specify CARRIAGECONTROL = 'NONE' when you open the file to be compressed.

3. Create the compression/expansion function—Use the DCX$MAKE_MAP routine to create the compression/expansion function. You pass DCX$MAKE_MAP a context variable, and DCX$MAKE_MAP uses the information stored in the associated work area to compute a compression /expansion function for the records being compressed. If DCX$MAKE_MAP returns a status value of DCX$_AGAIN, repeat steps 2 and 3 until DCX$MAKE_MAP returns a status of DCX$_NORMAL indicating that a compression/expansion function has been created.

   In Example 8–10, the integer function GET_MAP analyzes each record in the file to be compressed and invokes DCX$MAKE_MAP to create the compression/expansion function. The function value of GET_MAP is the return status of DCX$MAKE_MAP, and the address and length of the compression/expansion function are returned in GET_MAP's argument list. The main program, COMPRESS, invokes the GET_MAP function, examines its function value, and, if necessary, invokes GET_MAP again (see the ANALYZE DATA section of COMPRESS.FOR).

4. Clean up the analysis work area—Use the DCX$ANALYZE_DONE routine to delete a work area. Identify the work area to be deleted by passing DCX$ANALYZE_DONE a context variable.

5. Save the compression/expansion function—You cannot expand compressed records without the compression/expansion function. Therefore, before compressing the records, write the compression /expansion function to the file that will contain the compressed records.

   If your programming language cannot use an address directly, pass the address of the compression/expansion function to a subprogram (WRITE_MAP in the following example). Pass the subprogram the length of the compression/expansion function as well.

In the subprogram, declare the dummy argument corresponding to the function address as a one-dimensional, adjustable, byte array. Declare the dummy argument corresponding to the function length as an integer and use it to dimension the adjustable array. Write the function length and the array containing the function to the file that is to contain the compressed records. (The length must be stored so that you can read the function from the file using unformatted I/O; see Section 8.5.2.)

6 Compress each record—Use the DCX$COMPRESS_INIT routine to initialize a compression work area. Specify a context variable for the compression area just as for the analysis area.

Use the DCX$COMPRESS_DATA routine to compress each record. As you compress each record, use unformatted I/O to write the compressed record to the file containing the compression/expansion function. For each record, write the length of the record and the substring containing the record. See the COMPRESS DATA section in the following example. (The length is stored with the substring so that you can read the compressed record from the file using unformatted I/O; see Section 8.5.2.)

Use DCX$COMPRESS_DONE to delete the work area created by DCX$COMPRESS_INIT. Identify the work area to be deleted by passing DCX$COMPRESS_DATA a context variable. Use LIB$FREE_VM to free the virtual memory that DCX$MAKE_MAP used for the compression /expansion function.

Example 8–10 Compressing Data

```
!COMPRESS.FOR
    .
    .
    .
! Status variable
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'
EXTERNAL DCX$_AGAIN
! Context variable
INTEGER CONTEXT
! Compression/expansion function
INTEGER MAP,
2       MAP_LEN
! Normal file name, length, and logical unit number
CHARACTER*256 NORM_NAME
INTEGER*2 NORM_LEN
INTEGER NORM_LUN
! Compressed file name, length, and logical unit number
CHARACTER*256 COMP_NAME
INTEGER*2 COMP_LEN
INTEGER COMP_LUN
```

Example 8–10 Cont'd. on next page

Example 8–10 (Cont.) Compressing Data

```
! Logical end-of-file
LOGICAL EOF
! Record buffers;  32767 is maximum record size
CHARACTER*32767 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN
! User routine
INTEGER GET_MAP,
2       WRITE_MAP
! Library procedures
INTEGER DCX$ANALYZE_INIT,
2       DCX$ANALYZE_DONE,
2       DCX$COMPRESS_INIT,
2       DCX$COMPRESS_DATA,
2       DCX$COMPRESS_DONE,
2       LIB$GET_INPUT,
2       LIB$GET_LUN,
2       LIB$FREE_VM
! Get name of file to be compressed and open it
STATUS = LIB$GET_INPUT (NORM_NAME,
2                       'File to compress: ',
2                       NORM_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_LUN (NORM_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = NORM_LUN,
2     FILE = NORM_NAME(1:NORM_LEN),
2     CARRIAGECONTROL = 'NONE',
2     STATUS = 'OLD')
! ************
! ANALYZE DATA
! ************
! Initialize work area
STATUS = DCX$ANALYZE_INIT (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get compression/expansion function (map)
STATUS = GET_MAP (NORM_LUN,
2                 CONTEXT,
2                 MAP,
2                 MAP_LEN)
DO WHILE (STATUS .EQ. %LOC(DCX$_AGAIN))
  ! Go back to beginning of file
  REWIND (UNIT = NORM_LUN)
  ! Try map again
  STATUS = GET_MAP (NORM_LUN,
2                   CONTEXT,
2                   MAP,
2                   MAP_LEN)
END DO
```

Example 8–10 Cont'd. on next page

Example 8–10 (Cont.) Compressing Data

```
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Clean up work area
STATUS = DCX$ANALYZE_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! *************
! COMPRESS DATA
! *************
! Go back to beginning of file to be compressed
REWIND (UNIT = NORM_LUN)
! Open file to hold compressed records
STATUS = LIB$GET_LUN (COMP_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (COMP_NAME,
2                       'File for compressed records: ',
2                       COMP_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = COMP_LUN,
2     FILE = COMP_NAME(1:COMP_LEN),
2     STATUS = 'NEW',
2     FORM = 'UNFORMATTED')
! Initialize work area
STATUS = DCX$COMPRESS_INIT (CONTEXT,
2                           MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Write compression/expansion function to new file
CALL WRITE_MAP (COMP_LUN,
2               %VAL(MAP),
2               MAP_LEN)
! Read record from file to be compressed
EOF = .FALSE.
READ (UNIT = NORM_LUN,
2     FMT = '(Q,A)',
2     IOSTAT = IOSTAT) RECORD_LEN,
2                      RECORD(1:RECORD_LEN)
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
     CALL LIB$SIGNAL (%VAL(STATUS))
  ELSE
     EOF = .TRUE.
     STATUS = STATUS_OK
  END IF
END IF
DO WHILE (.NOT. EOF)
  ! Compress the record
  STATUS = DCX$COMPRESS_DATA (CONTEXT,
2                             RECORD(1:RECORD_LEN),
2                             RECORD2,
2                             RECORD2_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  ! Write compressed record to new file
  WRITE (UNIT = COMP_LUN) RECORD2_LEN
  WRITE (UNIT = COMP_LUN) RECORD2 (1:RECORD2_LEN)
```

Example 8–10 Cont'd. on next page

Example 8–10 (Cont.) Compressing Data

```
  ! Read from file to be compressed
  READ (UNIT = NORM_LUN,
2       FMT = '(Q,A)',
2       IOSTAT = IOSTAT) RECORD_LEN,
2                        RECORD (1:RECORD_LEN)
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
       CALL LIB$SIGNAL (%VAL(STATUS))
    ELSE
       EOF = .TRUE.
       STATUS = STATUS_OK
    END IF
  END IF
END DO
! Close files and clean up work area
CLOSE (NORM_LUN)
CLOSE (COMP_LUN)
STATUS = LIB$FREE_VM (MAP_LEN,
2                     MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = DCX$COMPRESS_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

GET_MAP.FOR

```
INTEGER FUNCTION GET_MAP (LUN,       ! Passed
2                         CONTEXT,   ! Passed
2                         MAP,       ! Returned
2                         MAP_LEN)   ! Returned
! Analyzes records in file opened on logical
! unit LUN and then attempts to create a
! compression/expansion function using
! DCX$MAKE_MAP.
! Dummy arguments
! Context variable
INTEGER CONTEXT
! Logical unit number
INTEGER LUN
! Compression/expansion function
INTEGER MAP,
2       MAP_LEN
! Status variable
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'
! Logical end-of-file
LOGICAL EOF
! Record buffer; 32767 is the maximum record size
CHARACTER*32767 RECORD
INTEGER RECORD_LEN
```

Example 8–10 Cont'd. on next page

Example 8–10 (Cont.) Compressing Data

```
! Library procedures
INTEGER DCX$ANALYZE_DATA,
2       DCX$MAKE_MAP
! Analyze records
EOF = .FALSE.
READ (UNIT = LUN,
2     FMT = '(Q,A)',
2     IOSTAT = IOSTAT) RECORD_LEN,RECORD
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
     CALL LIB$SIGNAL (%VAL(STATUS))
  ELSE
     EOF = .TRUE.
     STATUS = STATUS_OK
  END IF
END IF
DO WHILE (.NOT. EOF)
  STATUS = DCX$ANALYZE_DATA (CONTEXT,
2                           RECORD(1:RECORD_LEN))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  READ (UNIT = LUN,
2       FMT = '(Q,A)',
2       IOSTAT = IOSTAT) RECORD_LEN,RECORD
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
       CALL LIB$SIGNAL (%VAL(STATUS))
    ELSE
       EOF = .TRUE.
       STATUS = STATUS_OK
    END IF
  END IF
END DO

STATUS = DCX$MAKE_MAP (CONTEXT,
2                      MAP,
2                      MAP_LEN)
GET_MAP = STATUS

END
```

WRITE_MAP.FOR

```
SUBROUTINE WRITE_MAP (LUN,     ! Passed
2                     MAP,     ! Passed
2                     MAP_LEN) ! Passed
! Write compression/expansion function
! to file of compressed data
! Dummy arguments
INTEGER LUN,         ! Logical unit of file
2       MAP_LEN      ! Length of function
BYTE MAP (MAP_LEN)   ! Compression/expansion function
```

Example 8–10 Cont'd. on next page

Example 8–10 (Cont.) Compressing Data

```
! Write map length
WRITE (UNIT = LUN) MAP_LEN
! Write map
WRITE (UNIT = LUN) MAP

END
```

8.5.2 Expansion Routines

Perform the following steps to expand a compressed file:

1. Read the compression/expansion function—When reading the compression/expansion function from the compressed file, do not make any assumptions about the function's size. The best practice is to read the length of the function from the compressed file and then invoke the LIB$GET_VM routine to get the necessary amount of storage for the function. LIB$GET_VM returns the address of the first byte of the storage area.

   If your programming language cannot use an address directly, pass the address of the storage area to a subprogram (READ_MAP in the following example). Pass the subprogram the length of the compression /expansion function as well.

   In the subprogram, declare the dummy argument corresponding to the storage address as a one-dimensional, adjustable, BYTE array. Declare the dummy argument corresponding to the function length as an integer and use it to dimension the adjustable array. Read the compression /expansion function from the compressed file into the dummy array. Since the compression/expansion function is stored in the subprogram, do not return to the main program until you have expanded all of the compressed records.

2. Initialize an expansion work area—Use the DCX$EXPAND_INIT routine to initialize a work area for expanding the records. The first argument passed to DCX$EXPAND_INIT is an integer variable to contain a context value (see step 1 in Section 8.5.1). The second argument is the address of the compression/expansion function.

3. Expand the records—Use the DCX$EXPAND_DATA routine to expand each record.

4. Clean up the work area—Use the DCX$EXPAND_DONE routine to delete an expansion work area. Identify the work area to be deleted by passing DCX$EXPAND_DONE a context variable.

Example 8–11 expands a compressed file. The first record of the compressed file is an integer containing the number of bytes in the compression /expansion function. The second record is the compression/expansion function. The remainder of the file contains the compressed records. Each compressed record is stored as two records, an integer containing the length of the record and a substring containing the record.

Example 8–11 Expanding Data

```
!EXPAND.FOR
     .
     .
     .
INTEGER STATUS

! File names, lengths, and logical unit numbers
CHARACTER*256 OLD_FILE,
2             NEW_FILE
INTEGER*2 OLD_LEN,
2         NEW_LEN
INTEGER OLD_LUN,
2       NEW_LUN
! Length of compression/expansion function
INTEGER MAP,
2       MAP_LEN
! User routine
EXTERNAL EXPAND_DATA
! Library procedures
INTEGER LIB$GET_LUN,
2       LIB$GET_INPUT,
2       LIB$GET_VM,
2       LIB$FREE_VM
! Open file to expand
STATUS = LIB$GET_LUN (OLD_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (OLD_FILE,
2                       'File to expand: ',
2                       OLD_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = OLD_LUN,
2     STATUS = 'OLD',
2     FILE = OLD_FILE(1:OLD_LEN),
2     FORM = 'UNFORMATTED')
! Open file to hold expanded data
STATUS = LIB$GET_LUN (NEW_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$GET_INPUT (NEW_FILE,
2                       'File to hold expanded data: ',
2                       NEW_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = NEW_LUN,
2     STATUS = 'NEW',
2     CARRIAGECONTROL = 'NONE',
2     FILE = NEW_FILE(1:NEW_LEN))
```

Example 8–11 Cont'd. on next page

Example 8–11 (Cont.) Expanding Data

```
! Expand file
! Get length of compression/expansion function
READ (UNIT = OLD_LUN) MAP_LEN
STATUS = LIB$GET_VM (MAP_LEN,
2                    MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Expand records
CALL EXPAND_DATA (%VAL(MAP),
2                 MAP_LEN,      ! Length of function
2                 OLD_LUN,      ! Compressed data file
2                 NEW_LUN)      ! Expanded data file
! Delete virtual memory used for function
STATUS = LIB$FREE_VM (MAP_LEN,
2                     MAP)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
END
```

EXPAND_DATA.FOR

```
SUBROUTINE EXPAND_DATA (MAP,      ! Passed
2                       MAP_LEN, ! Passed
2                       OLD_LUN, ! Passed
2                       NEW_LUN) ! Passed
! Expand data program
! Dummy arguments
INTEGER MAP_LEN,   ! Length of expansion function
2       OLD_LUN,   ! Logical unit of compressed file
2       NEW_LUN    ! logical unit of expanded file
BYTE MAP(MAP_LEN)  ! Array containing the function
! Status variables
INTEGER STATUS,
2       IOSTAT,
2       IO_OK,
2       STATUS_OK
PARAMETER (IO_OK = 0)
PARAMETER (STATUS_OK = 1)
INCLUDE '($FORDEF)'
! Context variable
INTEGER CONTEXT
! Logical end-of-file
LOGICAL EOF
! Record buffers
CHARACTER*32767 RECORD,
2               RECORD2
INTEGER RECORD_LEN,
2       RECORD2_LEN
! Library procedures
INTEGER DCX$EXPAND_INIT,
2       DCX$EXPAND_DATA,
2       DCX$EXPAND_DONE
```

Example 8–11 Cont'd. on next page

Example 8–11 (Cont.) Expanding Data

```
! Read data compression/expansion function
READ (UNIT = OLD_LUN) MAP
! Initialize work area
STATUS = DCX$EXPAND_INIT (CONTEXT,
2                        %LOC(MAP(1)))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Expand records
EOF = .FALSE.
! Read length of compressed record
READ (UNIT = OLD_LUN,
2     IOSTAT = IOSTAT) RECORD_LEN
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ENDDURREA) THEN
    CALL LIB$SIGNAL (%VAL(STATUS))
  ELSE
    EOF = .TRUE.
    STATUS = STATUS_OK
  END IF
END IF
DO WHILE (.NOT. EOF)
  ! Read compressed record
  READ (UNIT = OLD_LUN) RECORD (1:RECORD_LEN)
  ! Expand record
  STATUS = DCX$EXPAND_DATA (CONTEXT,
2                           RECORD(1:RECORD_LEN),
2                           RECORD2,
2                           RECORD2_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  ! Write expanded record to new file
  WRITE (UNIT = NEW_LUN,
2        FMT = '(A)') RECORD2(1:RECORD2_LEN)
  ! Read length of compressed record
  READ (UNIT = OLD_LUN,
2       IOSTAT = IOSTAT) RECORD_LEN
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL(STATUS))
    ELSE
      EOF = .TRUE.
      STATUS = STATUS_OK
    END IF
  END IF
END DO
! Clean up work area
STATUS = DCX$EXPAND_DONE (CONTEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

END
```

8.6 Librarian Utility Routines

You can use LIBRARIAN to manage libraries at the programming level by using Librarian Utility routines (LBR). The *VMS Utility Routines Manual* contains complete specifications for the routines and their arguments.

For more information on using LIBRARIAN from DCL, refer to Chapter 5 and the *VMS Librarian Utility Manual.*

8.6.1 Creating, Opening, and Closing Libraries

Using a library requires the following sequence of events:

1. Initialize the library—Call LBR$INI_CONTROL to initialize the library. LBR$INI_CONTROL returns a value to the first argument that you must use in the remaining calls to the LBR routines; do not alter this value. Pass one of the following values as the second argument: LBR$C_CREATE to create and update a new library; LBR$C_UPDATE to update an existing library; LBR$C_READ to read (no updates allowed) from an existing library.
2. Open the library—Call LBR$OPEN to open the library. Pass the value returned by LBR$INI_CONTROL as the first argument. Pass the file specification or partial file specification of the library file as the second argument and any defaults for the file specification as the fourth argument. (The current default device and directory are used if these parts of the file specification remain unspecified.) If you are creating a new library, pass the create options array as the third argument. The CRE$ symbols (see the specifications in the *VMS Utility Routines Manual*) identify the significant longwords of the array by their byte offsets into the array. Convert these values to subscripts for an array of integers (longwords) by dividing by 4 and adding 1. If you do not load the significant longwords before calling LBR$INI_CONTROL, the library may be corrupted upon creation.
3. Work with the library—Call the various LBR routines and perform other operations according to your program design.
4. Close the library—Call LBR$CLOSE to close the library. Supply the value returned by LBR$INI_CONTROL as the first and only argument. You must close a library explicitly for updates to be posted.

Note: Do not use LBR$INI_CONTROL, LBR$OPEN, and LBR$CLOSE for writing help text with LBR$OUTPUT_HELP. Simply invoke LBR$OUTPUT_HELP.

Certain symbols used by the LBR routines are not defined in the default object and shareable image libraries. You must include them explicitly by calling $LBRDEF, $CREDEF, $MHDDEF, $LHIDEF, and $HLPDEF (as noted in the specifications in the *VMS Utility Routines Manual*) in macro programs specifying GLOBAL as an argument and by linking these programs with your application program.

To open a library if it exists, or to create and open it if it does not exist, try to open the library in UPDATE or READ mode, checking for an error status value of RMS$_FNF. If this error occurs, open the library in CREATE mode. Otherwise, open the library in UPDATE or READ mode. Example 8–12 opens, or creates and opens, a text library.

Example 8–12 Creating, Opening, and Closing a Text Library

```
!DOLIB.CLD
   .
   .
   .
! Defines the command to call DOLIB.EXE
DEFINE VERB DOLIB
IMAGE WORK:[TEXTLIB]DOLIB
! Specify the library name (not the full spec)
! Defaults to current directory and a file type of TLB
PARAMETER P1,LABEL=LIBSPEC,PROMPT="Library",VALUE(REQUIRED)
! Specify the action to be performed
QUALIFIER ENTER                        ! /ENTER
QUALIFIER EXTRACT, VALUE (LIST)        ! /EXTRACT=(module,...)
QUALIFIER DELETE, VALUE (LIST)         ! /DELETE=(module,...)
QUALIER TYPEINFO                      ! /TYPEINFO
QUALIFIER MODHEAD, VALUE (LIST)        ! /MODHEAD=(module,...)
QUALIFIER LIST, VALUE (DEFAULT="*")    ! /LIST[=matchname]
QUALIFIER ALIAS, VALUE (LIST)          ! /ALIAS=(module,alias,...)
QUALIFIER SHOWALIAS, VALUE (REQUIRED)  ! /SHOWALIAS=module
```

DOLIBMSG.MSG

```
 .TITLE DOLIB messages
 .FACILITY DOLIB, 1 /PREFIX=DOLIB_
 .SEVERITY WARNING
MODEX "Module already exists --- '!AS'" /FAO=1
NOMOD "No such module --- '!AS'" /FAO=1
 .SEVERITY SEVERE
NOACTION "No action specified on command line"
 .END
```

LBRDEF.MAR

```
 $LBRDEF   GLOBAL
 $CREDEF   GLOBAL
 $MHDDEF   GLOBAL
 $LHIDEF   GLOBAL
 $HLPDEF   GLOBAL
 .END
```

DOLIB.FOR

```
PROGRAM DOLIB

! Implements user requests on text libraries
! Command line: DOLIB [qualifier] library-name
!   Qualifiers: /ENTER
!               /EXTRACT=(module-name,...)
!               /DELETE=(module-name,...)
!               /TYPEINFO
!               /MODHEAD=(module-name,...)
!               /LIST[=match-name]
!               /ALIAS=(module,alias,...)
!               /SHOWALIAS=module
```

Example 8–12 Cont'd. on next page

Example 8–12 (Cont.) Creating, Opening, and Closing a Text Library

```
CHARACTER*31 LIBSPEC,      ! Library file
2            STATUS,       ! Return status
2            INDEX,        ! Library index
2            FUNC,         ! Library function
2            OPTIONS (20), ! Create options
2            TYPE,         ! Subscripts for create options
2            KEYLEN,
2            ALLOC,
2            IDXMAX,
2            UHDMAX,
2            ENTALL
! VMS library procedures
INTEGER LBR$INI_CONTROL,
2       LBR$OPEN,
2       LBR$CLOSE,
2       CLI$PRESENT
! Offsets for create options array --- defined in $CREDEF
EXTERNAL CRE$L_TYPE,   ! Library type
2        CRE$L_KEYLEN, ! Maximum key length
2        CRE$L_ALLOC,  ! Initial allocation
2        CRE$L_IDXMAX, ! Number of indexes
2        CRE$L_UHDMAX, ! Module header extra bytes
2        CRE$L_ENTALL  ! Preallocated index entries
! Type and function values --- defined in $LBRDEF
EXTERNAL LBR$C_TYP_UNK, ! Unknown
2        LBR$C_TYP_OBJ, ! Object or shareable image
2        LBR$C_TYP_MLB, ! Macro
2        LBR$C_TYP_HLP, ! Help
2        LBR$C_TYP_TXT, ! Text
2        LBR$C_CREATE,  ! Create new library
2        LBR$C_READ,    ! Open for read only
2        LBR$C_UPDATE   ! Update
! Return codes
EXTERNAL RMS$_FNF,      ! File not found
2        DOLIB_NOACTION ! No action specified
! Get library name
CALL CLI$GET_VALUE ('LIBSPEC',
2                  LIBSPEC)
! Determine function --- update or read only
! Read only on /EXTRACT, /TYPEINFO, /LIST, /SHOWALIAS
IF (CLI$PRESENT ('EXTRACT') .OR.
2   CLI$PRESENT ('TYPEINFO') .OR.
2   CLI$PRESENT ('LIST') .OR.
2   CLI$PRESENT ('SHOWALIAS')) THEN
  FUNC = %LOC (LBR$C_READ)
ELSE
  FUNC = %LOC (LBR$C_UPDATE)
END IF

! Initialize and open library
STATUS = LBR$INI_CONTROL (INDEX,
2                         FUNC)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LBR$OPEN (INDEX,
2                  LIBSPEC,,
2                  '.TLB')
```

Example 8–12 Cont'd. on next page

Example 8–12 (Cont.) Creating, Opening, and Closing a Text Library

```
! If library does not exist, create it
IF (STATUS .EQ. %LOC (RMS$_FNF)) THEN
  ! Initialize with function = create
  STATUS = LBR$INI_CONTROL (INDEX,
2                           %LOC (LBR$C_CREATE))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Calculate subscripts for create options array
  TYPE = %LOC (CRE$L_TYPE) / 4 + 1
  KEYLEN = %LOC (CRE$L_KEYLEN) / 4 + 1
  ALLOC = %LOC (CRE$L_ALLOC) / 4 + 1
  IDXMAX = %LOC (CRE$L_IDXMAX) / 4 + 1
  UHDMAX = %LOC (CRE$L_UHDMAX) / 4 + 1
  ENTALL = %LOC (CRE$L_ENTALL) / 4 + 1
  ! Load create options array
  OPTIONS (TYPE) = %LOC (LBR$C_TYP_TXT)
  OPTIONS (KEYLEN) = 31
  OPTIONS (ALLOC) = 8
  OPTIONS (IDXMAX) = 2
  OPTIONS (UHDMAX) = 64
  OPTIONS (ENTALL) = 96
  ! Open library
  STATUS = LBR$OPEN (INDEX,
2                    LIBSPEC,
2                    OPTIONS,
2                    '.TLB')
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
ELSE IF ((.NOT. STATUS) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Dispatch per user request
IF (CLI$PRESENT ('ENTER')) THEN
  CALL ENTER (INDEX)
ELSE IF (CLI$PRESENT ('EXTRACT')) THEN
  CALL EXTRACT (INDEX)
ELSE IF (CLI$PRESENT ('DELETE')) THEN
  CALL DELETE (INDEX)
ELSE IF (CLI$PRESENT ('TYPEINFO')) THEN
  CALL TYPEINFO (INDEX)
ELSE IF (CLI$PRESENT ('MODHEAD')) THEN
  CALL MODHEAD (INDEX)
ELSE IF (CLI$PRESENT ('LIST')) THEN
  CALL LIST (INDEX)
ELSE IF (CLI$PRESENT ('ALIAS')) THEN
  CALL ALIAS (INDEX)
ELSE IF (CLI$PRESENT ('SHOWALIAS')) THEN
  CALL SHOWAL (INDEX)
ELSE
  CALL LIB$SIGNAL (%LOC (DOLIB_NOACTION))
END IF
! Close library
STATUS = LBR$CLOSE (INDEX)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! Exit
END
```

8.6.2 Adding Modules

Use the following routines to insert new modules into an open library:

1. LBR$LOOKUP_KEY (optional)—Ensure that the module does not already exist by calling LBR$LOOKUP_KEY. The expected return status is LBR$_KEYNOTFND.
2. LBR$PUT_RECORD—Construct the module by calling LBR$PUT_RECORD once for each record going into the module. Pass the contents of the record as the second argument. LBR$PUT_RECORD returns the record file address (RFA) in the library file as the third argument on the first call. On subsequent calls, you pass the RFA as the third argument, so do not alter its value between calls.
3. LBR$PUT_END—Call LBR$PUT_END after the last call to LBR$PUT_RECORD.
4. LBR$INSERT_KEY—Call LBR$INSERT_KEY to catalog the records you have just put in the library. The second argument is the name of the module.

To replace an existing module, save the old RFA returned by LBR$LOOKUP_KEY—step 1 above—(you should not receive an error message) in one variable and the new RFA returned by the first call to LBR$PUT_RECORD (step 2) in another variable. On step 4, invoke LBR$REPLACE_KEY instead of LBR$INSERT_KEY, pass the old RFA as the third argument, and the new RFA as the fourth argument.

The subroutine in Example 8–13 solicits module names and text from SYS$INPUT and adds modules to a text library.

Example 8–13 Adding Modules to a Text Library

```
    .
    .
    .
SUBROUTINE ENTER (INDEX)
! Enters text modules into library from SYS$INPUT

INTEGER STATUS,         ! Return status
2       INDEX,          ! Library index
2       TXTRFA (2)      ! RFA of module
CHARACTER*31 MODNAME    ! Name of module
CHARACTER*255 TEXTLINE  ! One record of text
INTEGER*2 MODNAME_LEN,  ! Length of module name
2         TEXTLINE_LEN  ! Length of text record
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$PUT_RECORD,
2       LBR$PUT_END,
2       LBR$INSERT_KEY,
2       LBR$GET_INPUT,
2       LIB$PUT_OUTPUT
! Return codes
EXTERNAL RMS$_EOF,        ! End-of-file
2        LBR$_KEYNOTFND,  ! Key not found
2        DOLIB_MODEX      ! Module already exists
```

Example 8–13 Cont'd. on next page

Example 8–13 (Cont.) Adding Modules to a Text Library

```
! Get first module name
STATUS = LIB$GET_INPUT (MODNAME,
2                       'Module name or CTRL/Z: ',
2                       MODNAME_LEN)
IF ((.NOT. STATUS) .AND.
2   (STATUS .NE. %LOC (RMS$_EOF))) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Insert modules until end-of-file
DO WHILE (STATUS .NE. %LOC (RMS$_EOF))
  ! Verify that module does not already exist
  STATUS = LBR$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
  ! Insert module in library
  IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
    ! Get first line of text
    STATUS = LIB$PUT_OUTPUT
2   ('Enter lines of text. Terminate with CTRL/Z:')
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$GET_INPUT (TEXTLINE,,
2                           TEXTLINE_LEN)
    IF ((.NOT. STATUS) .AND.
2       (STATUS .NE. %LOC (RMS$_EOF))) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
    ! Insert text lines until end-of-file
    DO WHILE (STATUS .NE. %LOC (RMS$_EOF))
      ! Insert line
      STATUS = LBR$PUT_RECORD (INDEX,
2                              TEXTLINE (1:TEXTLINE_LEN),
2                              TXTRFA)
      IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
      ! Get another line
      STATUS = LIB$GET_INPUT (TEXTLINE,, TEXTLINE_LEN)
      IF ((.NOT. STATUS) .AND.
2         (STATUS .NE. %LOC (RMS$_EOF))) THEN
        CALL LIB$SIGNAL (%VAL (STATUS))
      END IF
    END DO
    ! Terminate text and catalog module
    STATUS = LBR$PUT_END (INDEX)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LBR$INSERT_KEY (INDEX,
2                            MODNAME (1:MODNAME_LEN),
2                            TXTRFA)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  ! If module already exists
  ELSE IF (STATUS) THEN
    CALL LIB$SIGNAL (DOLIB_MODEX,
2                    %VAL (1),
2                    MODNAME (1:MODNAME_LEN))
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
```

Example 8–13 Cont'd. on next page

Example 8–13 (Cont.) Adding Modules to a Text Library

```
  ! Get another module name
  STATUS = LIB$GET_INPUT (MODNAME,
2                         'Module name: ',
2                         MODNAME_LEN)
  IF ((.NOT. STATUS) .AND.
2     (STATUS .NE. %LOC (RMS$_EOF))) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END DO

! Exit
END
```

8.6.3 Deleting Modules

Use the following routines to delete modules from a library:

1. LBR$LOOKUP_KEY—Call LBR$LOOKUP_KEY to locate the module. Specify the name of the module as the second argument. LBR$LOOKUP_KEY returns the RFA of the module as the third argument; do not alter this value.
2. LBR$DELETE_KEY—Call LBR$DELETE_KEY to delete the key for the module. Specify the name of the module as the second argument.
3. LBR$DELETE_DATA—Call LBR$DELETE_DATA to delete the module itself. Specify the RFA of the module as the second argument.

The subroutine in Example 8–14 gets module names from the command line and deletes the specified modules from a text library:

Example 8–14 Deleting Modules from a Text Library

```
   .
   .
   .
SUBROUTINE DELETE (INDEX)
! Deletes text modules named by the
! qualifier /DELETE=(module-name,...)
INTEGER STATUS,      ! Return status
2       INDEX,       ! Library index
2       TXTRFA (2)   ! RFA of module
CHARACTER*31 MODNAME ! Name of module
INTEGER MODNAME_LEN  ! Length of module name
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$DELETE_KEY,
2       LBR$DELETE_DATA,
2       CLI$GET_VALUE
2       LIB$LOCC
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
2        DOLIB_NOMOD     ! No such module
! Get module name from /DELETE on command line
STATUS = CLI$GET_VALUE ('DELETE', MODNAME)
```

Example 8–14 Cont'd. on next page

Example 8–14 (Cont.) Deleting Modules from a Text Library

```
! Delete modules until bad return status,
! which indicates end of qualifier values
DO WHILE (STATUS)
  ! Calculate length of module name
  MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
  ! Look up module name in library index
  STATUS = LBR$LOOKUP_KEY (INDEX,
2                         MODNAME (1:MODNAME_LEN),
2                         TXTRFA)
  ! Delete module if it exists
  IF (STATUS) THEN
    STATUS = LBR$DELETE_KEY (INDEX,
2                           MODNAME (1:MODNAME_LEN))
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LBR$DELETE_DATA (INDEX, TXTRFA)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Issue warning if it does not exist
    ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
    CALL LIB$SIGNAL (DOLIB_NOMOD,
2                   %VAL (1),
2                   MODNAME (1:MODNAME_LEN))
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Get another module name
  STATUS = CLI$GET_VALUE ('DELETE', MODNAME)
END DO

! Exit
END
```

8.6.4 Extracting Modules

Use the following routines to extract modules from a library:

1. LBR$LOOKUP_KEY—Call LBR$LOOKUP_KEY to locate the module. Specify the name of the module as the second argument. LBR$LOOKUP_KEY returns the RFA of the module as the third argument; do not alter this value.
2. LBR$GET_RECORD—Call LBR$GET_RECORD once for each record in the module. Specify a character string to receive the extracted record as the second argument. LBR$GET_RECORD returns a status value of RMS$_EOF after the last record in the module is extracted.

The subroutine in Example 8–15 gets module names from the command line, extracts the contents of the modules, and writes the contents to SYS$OUTPUT.

Example 8–15 Extracting Modules from a Text Library

```
   .
   .
   .
SUBROUTINE EXTRACT (INDEX)
! Extracts text modules named by the
! qualifier /EXTRACT=(module-name,...)
! and types their contents to SYS$OUTPUT
INTEGER STATUS,        ! Return status
2       INDEX,         ! Library index
2       TXTRFA (2)     ! RFA of module
CHARACTER*31 MODNAME   ! Name of module
CHARACTER*255 TEXTLINE ! Line of text
INTEGER MODNAME_LEN    ! Length of module name
INTEGER TEXTLINE_LEN   ! Length of line of text
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$GET_RECORD,
2       LIB$PUT_OUTPUT,
2       CLI$GET_VALUE,
2       LIB$LOCC
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
2        RMS$_EOF,       ! End of text in module
2        DOLIB_NOMOD     ! No such module
! Get module name from /EXTRACT on command line
STATUS = CLI$GET_VALUE ('EXTRACT', MODNAME)
! Extract modules until bad return status,
! which indicates end of qualifier values
DO WHILE (STATUS)
  ! Calculate length of module name
  MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
  ! Look up module name in library index
  STATUS = LBR$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
! Extract module if it exists
  IF (STATUS) THEN
    ! Get line of text
    STATUS = LBR$GET_RECORD (INDEX, TEXTLINE)
    IF ((.NOT. STATUS) .AND.
2       (STATUS .NE. %LOC (RMS$_EOF))) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
    ! Write and extract records until end-of-file
    DO WHILE (STATUS .NE. %LOC (RMS$_EOF))
      ! Calculate length of text
      TEXTLINE_LEN = 255
      DO WHILE ((TEXTLINE (TEXTLINE_LEN:TEXTLINE_LEN) .EQ. ' ')
2         .AND. (TEXTLINE_LEN .GT. 0))
        TEXTLINE_LEN = TEXTLINE_LEN - 1
      END DO
      ! Type text
      IF (TEXTLINE_LEN .GT. 0) THEN
        STATUS = LIB$PUT_OUTPUT (TEXTLINE (1:TEXTLINE_LEN))
      ELSE
        STATUS = LIB$PUT_OUTPUT (' ')
      END IF
      IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–15 Cont'd. on next page

Example 8–15 (Cont.) Extracting Modules from a Text Library

```
      ! Get another record
      TEXTLINE (1:255) = ' '
      STATUS = LBR$GET_RECORD (INDEX, TEXTLINE)
      IF ((.NOT. STATUS) .AND.
2         (STATUS .NE. %LOC (RMS$_EOF))) THEN
        CALL LIB$SIGNAL (%VAL (STATUS))
      END IF
    END DO
  STATUS = LIB$PUT_OUTPUT ('***END OF MODULE***')

  ! Issue warning if module does not exist
  ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
    CALL LIB$SIGNAL (DOLIB_NOMOD,
2                    %VAL (1),
2                    MODNAME (1:MODNAME_LEN))
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF

  STATUS = CLI$GET_VALUE ('EXTRACT', MODNAME)
END DO

! Exit
END
```

8.6.5 Using Multiple Keys and Multiple Indexes

You can point at the same module with more than one key. The keys can be in the primary index (index 1) or alternate indexes (indexes 2 through 10). The best method is to reserve the primary index for module names. In system-defined object libraries, index 2 contains the global symbols defined by the various modules. The subroutine in Example 8–16 associates additional keys (which the routine calls aliases) with modules and stores these keys in index 2.

Example 8–16 Associating Keys with Modules

```
SUBROUTINE ALIAS (INDEX)
! Catalogs modules by alias

INTEGER STATUS,         ! Return status
2       INDEX,          ! Library index
2       TXTRFA (2)      ! RFA of module
CHARACTER*31 MODNAME,   ! Name of module
2            ALIASNAME  ! Name of alias
INTEGER MODNAME_LEN     ! Length of module name
INTEGER ALIASNAME_LEN   ! Length of alias name
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$SET_INDEX,
2       LBR$INSERT_KEY,
2       LIB$GET_INPUT,
2       LIB$GET_VALUE
2       LIB$LOCC
```

Example 8–16 Cont'd. on next page

Example 8–16 (Cont.) Associating Keys with Modules

```
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
2        LBR$_DUPKEY,    ! Duplicate key
2        RMS$_EOF,       ! End of text in module
2        DOLIB_NOMOD     ! No such module
! Get module name from /ALIAS on command line
CALL CLI$GET_VALUE ('ALIAS', MODNAME)
! Calculate length of module name
MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
! Look up module name in library index
STATUS = LBR$LOOKUP_KEY (INDEX,
2                        MODNAME (1:MODNAME_LEN),
3                        TXTRFA)
END IF
! Insert aliases if module exists
IF (STATUS) THEN
  ! Set to index 2
  STATUS = LBR$SET_INDEX (INDEX, 2)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Get alias name from /ALIAS on command line
  STATUS = CLI$GET_VALUE ('ALIAS', ALIASNAME)
  ! Insert aliases in index 2 until bad return status
  ! which indicates end of qualifier values
  DO WHILE (STATUS)
    ! Calculate length of alias name
    ALIASNAME_LEN = LIB$LOCC (' ', ALIASNAME) - 1
    ! Put alias name in index
    STATUS = LBR$INSERT_KEY (INDEX,
2                            ALIASNAME (1:ALIASNAME_LEN),
2                            TXTRFA)
    IF ((.NOT. STATUS) .AND.
2       (STATUS .NE. %LOC (LBR$_DUPKEY)) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
    ! Get another alias
    STATUS = CLI$GET_VALUE ('ALIAS', ALIASNAME)
  END DO

  ! Issue warning if module does not exist
ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
  CALL LIB$SIGNAL (DOLIB_NOMOD,
2                  %VAL (1),
2                  MODNAME (1:MODNAME_LEN))
ELSE
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF

! Exit
END
```

You can look up a module using any of the keys associated with it. The following code fragment checks index 2 for a key if the lookup in the primary index fails.

```
STATUS = LBR$SET_INDEX (INDEX, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LBR$LOOKUP_KEY (INDEX,
2                        MODNAME (1:MODNAME_LEN),
2                        TXTRFA)
IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
  STATUS = LBR$SET_INDEX (INDEX, 2)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  STATUS = LBR$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

You can identify the keys associated with a module in two ways: (1) by looking up the module (LBR$LOOKUP_KEY) using one of the keys and (2) by searching (LBR$SEARCH) applicable indexes for the keys. LBR$SEARCH calls a user-written routine each time it retrieves a key. The routine must be an integer function defined as external that returns a success (odd number) or failure (even number) status. LBR$SEARCH stops processing on a return status of failure. The subroutine in Example 8–17 lists the names of keys in index 2 (the aliases) that point to a module identified on the command line by its name in the primary index.

Example 8–17 Listing Keys Associated with a Module

```
      .
      .
      .
SUBROUTINE SHOWAL (INDEX)
! Lists aliases for a module

INTEGER STATUS,       ! Return status
2       INDEX,        ! Library index
2       TXTRFA (2)    ! RFA for module text
CHARACTER*31 MODNAME ! Name of module
INTEGER MODNAME_LEN  ! Length of module name
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$SEARCH,
2       LIB$LOCC
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
2        DOLIB_NOMOD     ! No such module
! Search routine
EXTERNAL SEARCH
INTEGER SEARCH
! Get module name and calculate length
CALL CLI$GET_VALUE ('SHOWALIAS', MODNAME)
MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
! Look up module in index 1
2 STATUS = LBR$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Example 8–17 Cont'd. on next page

Example 8–17 (Cont.) Listing Keys Associated with a Module

```
! Search for alias names in index 2
2 STATUS = LBR$SEARCH (INDEX,
2                      2,
2                      TXTRFA,
2                      SEARCH)

END
INTEGER FUNCTION SEARCH (ALIASNAME, RFA)
! Function called for each alias name pointing to MODNAME
! Displays the alias name
INTEGER STATUS_OK,         ! Good return status
2       RFA (2)            ! RFA of module
PARAMETER (STATUS_OK = 1)  ! Odd number
CHARACTER*(*) ALIASNAME    ! Name of module

! Display module name
TYPE *, MODNAME

! Exit
SEARCH = STATUS_OK
END
```

8.6.6 Accessing Module Headers

You can store user information in the header of each module up to the amount specified at library creation time in the CRE$L_UHDMAX option. The total size of each header in bytes is the value of MHD$B_USRDAT (defined by the macro $MHDDEF—currently this value is 16) plus the value assigned to the CRE$L_UHDMAX option.

To put user data into a module header, first locate the module with LBR$LOOKUP_KEY; then move the data to the module header by invoking LBR$SET_MODULE, specifying the first argument (index value returned by LBR$INI_CONTROL), the second argument (RFA returned by LBR$LOOKUP_KEY), and the fifth argument (character string containing the user data).

To read user data from a module header, first locate the module with LBR$LOOKUP_KEY; then, retrieve the entire module header by invoking LBR$SET_MODULE, specifying the first, second, third (character string to receive the contents of the module header), and fourth (length of the module header) arguments. The user data starts at the byte offset defined by MHD$B_USRDAT. Convert this value to a character string subscript by adding 1.

Example 8–18 displays the user data portion of module headers on SYS$OUTPUT and applies updates from SYS$INPUT.

Example 8–18 Displaying the Module Header

```
   .
   .
   .
SUBROUTINE MODHEAD (INDEX)
! Modifies module headers

INTEGER STATUS,          ! Return status
2       INDEX,           ! Library index
2       TXTRFA (2)       ! RFA of module
CHARACTER*31 MODNAME     ! Name of module
INTEGER MODNAME_LEN      ! Length of module name
CHARACTER*80 HEADER      ! Module header
INTEGER HEADER_LEN       ! Length of module header
INTEGER USER_START       ! Start of user data in header
CHARACTER*64 USERDATA    ! User data part of header
INTEGER*2 USERDATA_LEN   ! Length of user data
! VMS library procedures
INTEGER LBR$LOOKUP_KEY,
2       LBR$SET_MODULE,
2       LIB$GET_INPUT,
2       LIB$PUT_OUTPUT,
2       CLI$GET_VALUE,
2       LIB$LOCC
! Offset to user data --- defined in $MHDDEF
EXTERNAL MHD$B_USRDAT
! Return codes
EXTERNAL LBR$_KEYNOTFND, ! Key not found
2        DOLIB_NOMOD     ! No such module
! Calculate start of user data in header
USER_START = %LOC (MHD$B_USRDAT) + 1
! Get module name from /MODHEAD on command line
STATUS = CLI$GET_VALUE ('MODHEAD', MODNAME)

! Get module headers until bad return status
! which indicates end of qualifier values
DO WHILE (STATUS)

  ! Calculate length of module name
  MODNAME_LEN = LIB$LOCC (' ', MODNAME) - 1
  ! Look up module name in library index
  STATUS = LBR$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
```

Example 8–18 Cont'd. on next page

Example 8–18 (Cont.) Displaying the Module Header

```
  ! Get header if module exists
  IF (STATUS) THEN
    STATUS = LBR$SET_MODULE (INDEX,
2                            TXTRFA,
2                            HEADER,
2                            HEADER_LEN)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Display header and solicit replacement
    STATUS = LIB$PUT_OUTPUT
2   ('User data for module '//MODNAME (1:MODNAME_LEN)//':')
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$PUT_OUTPUT
2   (HEADER (USER_START:HEADER_LEN))
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$PUT_OUTPUT
2   ('Enter replacement text below or just hit return:')
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = LIB$GET_INPUT (USERDATA,, USERDATA_LEN)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Replace user data
    IF (USERDATA_LEN .GT. 0) THEN
      STATUS = LBR$SET_MODULE (INDEX,
2                              TXTRFA,,,
2                              USERDATA (1:USERDATA_LEN))
    END IF

    ! Issue warning if module does not exist
  ELSE IF (STATUS .EQ. %LOC (LBR$_KEYNOTFND)) THEN
    CALL LIB$SIGNAL (DOLIB_NOMOD,
2                    %VAL (1),
2                    MODNAME (1:MODNAME_LEN))
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF

  ! Get another module name
  STATUS = CLI$GET_VALUE ('MODHEAD', MODNAME)
END DO

! Exit
END
```

8.6.7 Reading Library Headers

Call LBR$GET_HEADER to obtain general information concerning the library. Pass the value returned by LBR$INI_CONTROL as the first argument. LBR$GET_HEADER returns the information to the second argument, which must be an array of 128 longwords. The LHI$ symbols (see the specifications in the *VMS Utility Routines Manual*) identify the significant longwords of the array by their byte offsets into the array. Convert these values to subscripts by dividing by 4 and adding 1.

Example 8–19 reads the library header and displays some information from it.

Example 8–19 Reading Library Headers

```
   .
   .
   .
SUBROUTINE TYPEINFO (INDEX)
! Types the type, major ID, and minor ID
! of a library to SYS$OUTPUT

INTEGER STATUS              ! Return status
2       INDEX,              ! Library index
2       HEADER (128),       ! Structure for header information
2       TYPE,               ! Subscripts for header structure
2       MAJOR_ID,
2       MINOR_ID
CHARACTER*8 MAJOR_ID_TEXT, ! Display info in character format
2           MINOR_ID_TEXT
! VMS library procedures
INTEGER LBR$GET_HEADER,
2       LIB$PUT_OUTPUT
! Offsets for header --- defined in $LHIDEF
EXTERNAL LHI$L_TYPE,
2        LHI$L_MAJORID,
2        LHI$L_MINORID
! Library type values --- defined in $LBRDEF
EXTERNAL LBR$C_TYP_OBJ,
2        LBR$C_TYP_MLB,
2        LBR$C_TYP_HLP,
2        LBR$C_TYP_TXT
! Get header information
STATUS = LBR$GET_HEADER (INDEX, HEADER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Calculate subscripts for header structure
TYPE = %LOC (LHI$L_TYPE) / 4 + 1
MAJOR_ID = %LOC (LHI$L_MAJORID) / 4 + 1
MINOR_ID = %LOC (LHI$L_MINORID) / 4 + 1
! Display library type
IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_OBJ)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: object')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_MLB)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: macro')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_HLP)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: help')
ELSE IF (HEADER (TYPE) .EQ. %LOC (LBR$C_TYP_TXT)) THEN
  STATUS = LIB$PUT_OUTPUT ('Library type: text')
ELSE
  STATUS = LIB$PUT_OUTPUT ('Library type: unknown')
END IF
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Convert and display major ID
WRITE (UNIT=MAJOR_ID_TEXT,
2      FMT='(I)') HEADER (MAJOR_ID)
STATUS = LIB$PUT_OUTPUT ('Major ID: '//MAJOR_ID_TEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Convert and display minor ID
WRITE (UNIT=MINOR_ID_TEXT,
2      FMT='(I)') HEADER (MINOR_ID)
STATUS = LIB$PUT_OUTPUT ('Minor ID: '//MINOR_ID_TEXT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Exit
END
```

8.6.8 Displaying Help Text

You can display text from a help library by invoking LBR$OUTPUT_HELP, specifying the first (the output routine), third (the keywords), and fourth (the name of the library) arguments. You must also specify the last argument if the fifth argument indicates prompting mode or is omitted. Remember, subprograms specified in an argument list must be declared as external. LIB$PUT_OUTPUT and LIB$GET_INPUT can be used for the first and last arguments. (If you use your own routines, make sure the argument lists are the same as for LIB$PUT_OUTPUT and LIB$GET_INPUT.) Do not call LBR$INI_CONTROL and LBR$OPEN before calling LBR$OUTPUT_HELP. Example 8–20 solicits keywords from SYS$INPUT and displays the text associated with those keywords on SYS$OUTPUT, thus inhibiting the prompting facility.

Example 8–20 Displaying Text from a Help Library

```
PROGRAM GET_HELP

! Prints help text from a help library
CHARACTER*31 LIBSPEC    ! Library name
CHARACTER*15 KEYWORD    ! Keyword in help library
INTEGER*2 LIBSPEC_LEN,  ! Length of name
2         KEYWORD_LEN   ! Length of keyword
INTEGER FLAGS,          ! Help flags
2       STATUS          ! Return status
! VMS library procedures
INTEGER LBR$OUTPUT_HELP,
2       LIB$GET_INPUT,
2       LIB$PUT_OUTPUT
EXTERNAL LIB$GET_INPUT,
2        LIB$PUT_OUTPUT
! Error codes
EXTERNAL RMS$_EOF,       ! End-of-file
2        LIB$_INPSTRTRU ! Input string truncated
! Flag values --- defined in $HLPDEF
EXTERNAL HLP$M_PROMPT,
2        HLP$M_PROCESS,
2        HLP$M_GROUP,
2        HLP$M_SYSTEM,
2        HLP$M_LIBLIST,
2        HLP$M_HELP
! Get library name
STATUS = LIB$GET_INPUT (LIBSPEC,
2                       'Library: ',
2                       LIBSPEC_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
IF (LIBSPEC_LEN .EQ. 0) THEN
  LIBSPEC = 'HELPLIB'
  LIBSPEC_LEN = 7
END IF
```

Example 8–20 Cont'd. on next page

Example 8–20 (Cont.) Displaying Text from a Help Library

```
! Set flags for no prompting
FLAGS = %LOC (HLP$_PROCESS) +
2       %LOC (HLP$_GROUP) +
2       %LOC (HLP$_SYSTEM)

! Get first keyword
STATUS = LIB$GET_INPUT (KEYWORD,
2                       'Keyword or CTRL/Z: ',
2                       KEYWORD_LEN)
IF ((.NOT. STATUS) .AND.
2   (STATUS .NE. %LOC (LIB$_INPSTRTRU)) .AND.
2   (STATUS .NE. %LOC (RMS$_EOF))) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
! Display text until end-of-file
DO WHILE (STATUS .NE. %LOC (RMS$_EOF))
  STATUS = LBR$OUTPUT_HELP (LIB$PUT_OUTPUT,,
2                           KEYWORD (1:KEYWORD_LEN),
2                           LIBSPEC (1:LIBSPEC_LEN),
2                           FLAGS,
2                           LIB$GET_INPUT)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Get another keyword
  STATUS = LIB$GET_INPUT (KEYWORD,
2                         'Keyword or CTRL/Z: ',
2                         KEYWORD_LEN)
  IF ((.NOT. STATUS) .AND.
2     (STATUS .NE. %LOC (LIB$_INPSTRTRU)) .AND.
2     (STATUS .NE. %LOC (RMS$_EOF))) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END DO

! Exit
END
```

8.6.9 Listing and Processing Index Entries

You can process index entries an entry at a time by invoking LBR$GET_INDEX. The fourth argument specifies a match name for the entry or entries in the index to be processed: you can include the asterisk and percent characters in the match name for generic processing—for example, MOD* means all entries whose names begin with MOD; MOD% means all entries whose names are four characters and begin with MOD; and the asterisk (*) means all entries.

The third argument names a user-written routine that is executed once for each index entry specified by the fourth argument. The routine must be a function declared as external that returns a success (odd number) or failure (even number) status. LBR$GET_INDEX processing stops on a return status of failure. Declare the first argument passed to the function as a passed-length character argument—this argument contains the name of the index entry. Declare the second argument as an integer array of two elements.

Example 8–21 obtains a match name from the command line and displays the names of the matching entries from index 1 (the index containing the names of the modules).

Example 8–21 Displaying Index Entries

```
SUBROUTINE LIST (INDEX)
! Lists modules in the library

INTEGER STATUS,         ! Return status
2       INDEX,          ! Library index
CHARACTER*31 MATCHNAME ! Name of module to list
INTEGER MATCHNAME_LEN  ! Length of match name
! VMS library procedures
INTEGER address LBR$GET_INDEX,
3       LIB$LOCC
! Match routine
INTEGER MATCH
EXTERNAL MATCH
! Get module name and calculate length
CALL CLI$GET_VALUE ('LIST', MATCHNAME)
MATCHNAME_LEN = LIB$LOCC (' ', MATCHNAME) - 1
! Call routine to display module names
STATUS = LBR$GET_INDEX (INDEX,
2                          1, ! Primary index
3                          MATCH,
4                          MATCHNAME (1:MATCHNAME_LEN))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

! Exit
END
INTEGER FUNCTION MATCH (MODNAME, RFA)
! Function called for each module matched by MATCHNAME
! Displays the module name
INTEGER STATUS_OK,          ! Good return status
2       RFA (2)             ! RFA of module name in index
PARAMETER (STATUS_OK = 1)  ! Odd value
CHARACTER*(*) MODNAME       ! Name of module

! Display the name
TYPE *, MODNAME ! Display module name

! Exit
MATCH = STATUS_OK
END
```

8.7 File Definition Language

The File Definition Language (FDL) commands and routines provide a means of defining file characteristics. Typically, you use FDL to perform the following operations:

- Specify file characteristics otherwise unavailable from your language.
- Examine or modify the file characteristics of an existing data file in order to improve program or system interaction with that file.

You cannot specify FDL attributes when you open a file using language statements. Instead, use FDL to create your data file, set the desired file characteristics, and close the file; then, use a language statement to reopen the file. Since the data file is closed between the time the FDL attributes are set and the time your program accesses the file, you cannot use FDL to specify run-time attributes (attributes that are ignored or deleted when the associated data file is closed).

8.7.1 Creating an FDL File

An FDL file is a specially formatted text file containing a series of FDL attributes. You can create an FDL file with any text editor; however, to ensure that the file is correctly formatted, the best practice is to use the FDL editor or create the FDL file from an existing data file.

8.7.1.1 Using the FDL Editor

To invoke the FDL editor, use the EDIT/FDL command. Use the editor interactively to create new FDL files or to modify existing FDL files. Use the editor either interactively or noninteractively to optimize an FDL file in order to improve program or system interaction with the associated data file.

Throughout an interactive editing session (Section 8.7.2.2 describes noninteractive use of the editor), the FDL editor displays available subcommands or appropriate attributes, each followed by a brief description, and prompts you for a response. In general, a prompt consists of a short question, the type of value required or the range of acceptable values, and the default answer in brackets. If the question has no default answer, a hyphen appears within the brackets ([-]); in this case, you must supply an answer (or use CTRL/Z to abort the current command) before EDIT/FDL will continue the editing session.

If you are using FDL to specify a particular file characteristic that is unavailable from your programming langauge, use the editor subcommands ADD, DELETE, and MODIFY to edit the appropriate attribute. If you are using FDL to improve program or system interaction with an existing data file, have the editor optimize the associated FDL file (see Section 8.7.2.2). If you are using FDL to optimize program or system interaction with a data file that you have not yet created, use the editor subcommand INVOKE to choose an appropriate script. A script is a series of questions pertaining to the planned data file. By analyzing your responses to the questions, the editor determines which characteristics are best suited to the file and creates an FDL file describing those characteristics.

8.7.1.2 Using the Characteristics of an Existing Data File

To create an FDL file that describes the characteristics of an existing data file, use the DCL command ANALYZE/RMS_FILE/FDL or the FDL utility routine FDL$GENERATE. ANALYZE/RMS_FILE/FDL examines the specified data file and creates an FDL file that describes the characteristics of that file. FDL$GENERATE examines the RMS structures (the FAB and the RAB) of the specified data file and creates an FDL file that describes those structures.

Typically, an FDL file created by ANALYZE/RMS_FILE/FDL differs slightly from an FDL file created by FDL$GENERATE. (For example, if a file was created with no initial storage allocation and has since been allocated 30 blocks, the file section's ALLOCATE attribute in an FDL file created by FDL$GENERATE is 0; the same attribute in an FDL file created by ANALYZE /RMS_FILE/FDL is 30.) The FDL editor can optimize an FDL file created by

ANALYZE/RMS_FILE/FDL; however, it cannot optimize an FDL file created by FDL$GENERATE.

The following command creates an FDL file INCOME.FDL, which describes the characteristics of the data file INCOME83.DAT:

```
$ ANALYZE/RMS_FILE/FDL=INCOME INCOME83.DAT
```

For complete specifications for the ANALYZE/RMS_FILE command, see the *VMS DCL Dictionary*.

The program segment described in Example 8–22 creates an FDL file, INCOME.FDL, which describes the RMS structures of the data file INCOME83.DAT. Since the addresses of the FAB and RAB are only available within a user-open routine, FDL$GENERATE can be invoked only from within a user-open routine. Section 8.8 describes user-open routines. The *VMS Utility Routines Manual* contains complete specifications for FDL$GENERATE.

Example 8–22 Creating an FDL File

```
!MAIN.FOR
INTEGER LUN

! User-open routine
INTEGER FDL
EXTERNAL FDL
    .
    .
    .
STATUS = LIB$GET_LUN (LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
OPEN (UNIT = LUN,
2     FILE = 'INCOME83.DAT',
2     STATUS = 'OLD',
2     USEROPEN = FDL)
    .
    .
    .
```

USER_OPEN.FOR

```
INTEGER FUNCTION FDL (FAB,
2                     RAB,
2                     LUN)
! Generates an FDL file
! Dummy arguments
BYTE FAB(*),
2    RAB(*)
INTEGER LUN
! Mask for FDL$GENERATE
INTEGER MASK
EXTERNAL FDL$V_FULL_OUTPUT
! Status and library routine
INTEGER STATUS,
2       FDL$GENERATE
```

Example 8–22 Cont'd. on next page

Example 8–22 (Cont.) Creating an FDL File

```
MASK = IBSET (MASK, %LOC(FDL$V_FULL_OUTPUT))
STATUS = FDL$GENERATE (MASK,
2                     %LOC(FAB),
2                     %LOC(RAB),
2                     'TEST.FDL',
2                     ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
     .
     .
     .
! Return user-open status
FDL = STATUS

END
```

8.7.2 Applying an FDL File to a Data File

Use an FDL file to specify the file characteristics of a new data file or modify the file characteristics of an existing data file. When modifying file characteristics, the system creates a new data file and then reads the records from the existing data file to the new data file.

8.7.2.1 Creating a New Data File

To create a data file using the characteristics specified by an FDL file, use the DCL command CREATE/FDL or the library routine FDL$CREATE. The following command creates an empty data file INCOME83.DAT using the file characteristics specified by the FDL file INCOME.FDL:

```
$ CREATE/FDL=INCOME.FDL INCOME83.DAT
```

For complete specifications for the CREATE/FDL command, see the description of the Create/FDL Utility in the *VMS File Definition Language Facility Manual.*

The following program segment creates an empty data file named INCOME83.DAT using the file characteristics specified by the FDL file INCOME.FDL. The *STATEMENT* variable contains the number of the last FDL statement processed by FDL$CREATE; this argument is useful for debugging an FDL file. The *VMS Utility Routines Manual* contains complete specifications for FDL$CREATE.

```
INTEGER STATEMENT
INTEGER STATUS,
2       FDL$CREATE

STATUS = FDL$CREATE ('INCOME.FDL',
2                    'INCOME83.DAT',
2                    ,,,,
2                    STATEMENT,
2                    ,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
     .
     .
     .
```

8.7.2.2 Modifying an Existing Data File

To change the characteristics of an existing data file to those specified by an FDL file, use the DCL command CONVERT/FDL. (For complete specifications for the CONVERT command, see the description of the Convert Utility in the *VAX/VMS Convert Reference Manual.*) The following command changes the characteristics of the data file INCOME83.DAT to agree with those specified by the FDL file INCOME.FDL. The modified file is written to NEWINCOME83.DAT. (To write the modified data to a file with the same name as the original file, specify the second parameter as an asterisk.)

```
$ CONVERT/FDL=INCOME INCOME83.DAT NEWINCOME83.DAT
```

Typically, you change the characteristics of an existing data file to improve program or system interaction with that file. Unless you are familiar with VMS RMS and the internal structure of the file, the best practice is to allow the system to optimize the data file for you, as described in the following steps:

1 Create an FDL file—Use the DCL command ANALYZE/RMS_FILE/FDL to create an FDL file that describes the existing data file. The following command creates the FDL file INCOME.FDL, which describes the file characteristics of the data file INCOME83.DAT:

```
$ ANALYZE/RMS_FILE/FDL=INCOME INCOME83.DAT
```

2 Optimize the FDL file—Use the FDL editor noninteractively to optimize the FDL file. The following command writes an optimized version of INCOME.FDL to NEWINCOME.FDL. (Since an FDL file created by FDL$GENERATE describes the RMS structures rather than the file itself, EDIT/FDL cannot optimize an FDL file created by FDL$GENERATE and, therefore, does not accept such a file as input to a noninteractive session.)

```
$ EDIT/FDL/NOINTERACTIVE/ANALYZE=INCOME NEWINCOME
```

3 Change the data file—Use the DCL command CONVERT/FDL to change the characteristics of the existing data file to those specified by the optimized FDL file. The following command changes the file characteristics of the data file INCOME83.DAT to agree with those specified by the FDL file NEWINCOME.FDL. The modified file is written to NEWINCOME83.DAT.

```
$ CONVERT/FDL=NEWINCOME INCOME83.DAT NEWINCOME83.DAT
```

8.8 User-Open Routines

A user-open routine gives you direct access to the FAB and RAB (the VMS RMS structures that define file characteristics). Use a user-open routine to specify file characteristics otherwise unavailable from your programming language.

When you specify a user-open routine, you open the file rather than allow the program to open the file for you. Before passing the FAB and RAB to your user-open routine, any default file characteristics and characteristics that can be specified by keywords in the programming language are set. Your user-open routine should not set or modify such file characteristics because the language might not be aware that you have set the characteristics and might not perform as expected.

8.8.1 Opening a File

Section 8.3.1.2 provides guidelines on opening a file with a user-open routine. This section provides an example of a VAX FORTRAN user-open routine.

8.8.1.1 Specifying USEROPEN

To open a file with a user-open routine, include the USEROPEN specifier in the VAX FORTRAN OPEN statement. The value of the USEROPEN specifier is the name of the routine (not a character string containing the name). Declare the user-open routine as an INTEGER*4 function. Since the user-open routine name is specified as an argument, it must be declared in an EXTERNAL statement. The following statement instructs VAX FORTRAN to open SECTION.DAT using the routine UFO_OPEN:

```
! Logical unit number
INTEGER LUN

! Declare user-open routine
INTEGER UFO_OPEN
EXTERNAL UFO_OPEN
      .
      .
      .
OPEN (UNIT = LUN,
2     FILE = 'SECTION.DAT',
2     STATUS = 'OLD',
2     USEROPEN = UFO_OPEN)
      .
      .
      .
```

8.8.1.2 Writing the User-Open Routine

Write a user-open routine as an INTEGER function that accepts three dummy arguments:

- FAB address—Declare this argument as a RECORD variable. Use the record structure FABDEF defined in the $FABDEF module of SYS$LIBRARY:FORSYSDEF.TLB.
- RAB address—Declare this argument as a RECORD variable. Use the record structure RABDEF defined in the $RABDEF module of SYS$LIBRARY:FORSYSDEF.TLB.
- Logical unit number—Declare this argument as an INTEGER.

A user-open routine must perform at least the following operations. In addition, before opening the file, a user-open routine usually adjusts one or more fields in the FAB or the RAB or in both.

- Opens the file—To open the file, invoke the SYS$OPEN system service if the file already exists, or the SYS$CREATE system service if the file is being created.
- Connects the file—Invoke the SYS$CONNECT system service to establish a record stream for I/O.
- Returns the status—To return the status, equate the return status of the SYS$OPEN or SYS$CREATE system service to the function value of the user-open routine.

The following user-open routine opens an existing file. The file to be opened is specified in the OPEN statement of the invoking program unit.

UFO_OPEN.FOR

```
INTEGER FUNCTION UFO_OPEN (FAB,
2                          RAB,
2                          LUN)

! Include RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN
! Declare status variable
INTEGER STATUS
! Declare system routines
INTEGER SYS$CREATE,
2       SYS$OPEN,
2       SYS$CONNECT
! Optional FAB and/or RAB modifications
   .
   .
   .
! Open file
STATUS = SYS$OPEN (FAB)
IF (STATUS)
2  STATUS = SYS$CONNECT (RAB)

! Return status of $OPEN or $CONNECT
UFO_OPEN = STATUS

END
```

8.8.1.3 Setting FAB and RAB Fields

Each field in the FAB and RAB is identified by a symbolic name, such as FAB$L_FOP. Where separate bits in a field represent different attributes, each bit offset is identified by a similar symbolic name, such as FAB$V_CTG. The first three letters identify the structure containing the field. The letter following the dollar sign indicates either the length of the field (B for byte, W for word, or L for longword) or that the name is a bit offset (V for bit) rather than a field. The letters following the underscore identify the attribute associated with the field or bit. The symbol FAB$L_FOP identifies the FAB options field, which is a longword in length; the symbol FAB$V_CTG identifies the contiguity bit within the options field.

The STRUCTURE definitions for the FAB and RAB are in the $FABDEF and $RABDEF modules of the library SYS$LIBRARY:FORSYSDEF.TLB. To use these definitions, do the following:

1. Include the modules in your program unit.
2. Declare RECORD variables for the FAB and the RAB.
3. Reference the various fields of the FAB and RAB using the symbolic name of the field.

The following user-open routine specifies that the blocks allocated for the file must be contiguous. To specify contiguity, you clear the best-try-contiguous bit (FAB$V_CBT) of the FAB$L_FOP field and set the contiguous bit (FAB$V_CTG) of the same field.

UFO_CONTIG.FOR

```
INTEGER FUNCTION UFO_CONTIG (FAB,
2                            RAB,
2                            LUN)

! Include RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN
! Declare status variable
INTEGER STATUS
! Declare system procedures
INTEGER SYS$CREATE,
2       SYS$CONNECT
! Clear contiguous-best-try bit and
! set contiguous bit in FAB options
FAB.FAB$L_FOP = IBCLR (FAB.FAB$L_FOP, FAB$V_CBT)
FAB.FAB$L_FOP = IBSET (FAB.FAB$L_FOP, FAB$V_CTG)
! Open file
STATUS = SYS$CREATE (FAB)
IF (STATUS) STATUS = SYS$CONNECT (RAB)

! Return status of open or connect
UFO_CONTIG = STATUS

END
```

9 Condition Handling

Run-time errors are hardware- or software-detected events, usually errors, that alter normal program execution. Examples of run-time errors are as follows:

- System errors—For example, specifying an invalid argument to a system-defined procedure
- Language-specific errors—For example, in VAX FORTRAN, a data type conversion error during an I/O operation
- Application specific errors—For example, attempting to use invalid data

When an error occurs, VMS either returns a condition code identifying the error to your program or signals the condition code (Section 9.1.3 describes signaling). If VMS signals the condition code, an error message is typically displayed and program execution continues or terminates depending on the severity of the error.

Both an error message and its associated condition code identify an error by the name of the facility that generated it and an abbreviation of the message text. Therefore, if your program displays an error message, you can identify the condition code that was signaled. For example, if your program displays the following error message, you know that the condition code SS$_NOPRIV was signaled:

```
%SYSTEM-F-NOPRIV, no privilege for attempted operation
```

The descriptions of the system routines in the *VMS System Services Volume* and the *VMS Run-Time Library Routines Volume* include lists of the condition codes that may be returned by the routine.

9.1 General Error Handling

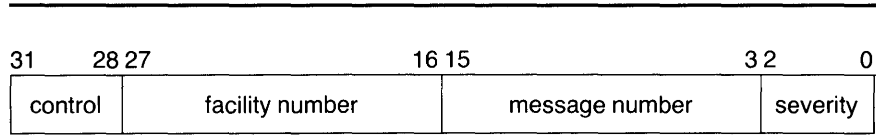
When unexpected errors occur, your program should display a message identifying the error, then either continue or stop, depending on the severity of the error. If you know that certain run-time errors might occur, you should provide special actions in your program to handle those errors.

9.1.1 Condition Code and Message

Error conditions are identified by integer values called condition codes. VMS defines condition codes to identify errors that might occur during execution of system-defined procedures. You can define condition codes for errors that might occur in your programs (see Section 9.2 for more information).

From a condition code you can determine whether any error has occurred, which particular error has occurred, and the severity of the error. Figure 9–1 illustrates the fields contained in a condition code.

Figure 9–1 Structure of a Condition Code



- Severity—The severity of the error condition. Bit <0> indicates success when set and failure when clear. Bits <1> and <2> distinguish degrees of success or failure. The three bits, when taken as an unsigned integer, are interpreted as shown in the following table. (The symbolic names are defined in module $STSDEF.)
- Message number—The number identifying the message associated with the error condition. The message may or may not be displayed when the associated error occurs.
- Facility number—The number identifying the facility (program) in which the error occurred. Bit <27> is set for user facilities and clear for DIGITAL facilities.
- Control—Control bits. Bit <28> inhibits the display of the error message; bits <31:29> are reserved for DIGITAL.

Code	Symbol	Severity	Response
0	STS$K_WARNING	Warning	Execution continues, unpredictable results
1	STS$K_SUCCESS	Success	Execution continues, expected results
2	STS$K_ERROR	Error	Execution continues, erroneous results
3	STS$K_INFO	Information	Execution continues, informational message displayed
4	STS$K_SEVERE	Severe error	Execution terminates, no output
5			Reserved for DIGITAL use only
6			Reserved for DIGITAL use only
7			Reserved for DIGITAL use only

9.1.2 Return Status Convention

Most system-defined procedures are functions of longwords, where the function value is equated to a condition code. In this capacity, the condition code is referred to as a return status. You can write your own routines to follow this convention. Each routine description in the *VMS System Services*

Volume and *VMS Run-Time Library Routines Volume* lists the condition codes that may be returned by that procedure.

9.1.2.1 Testing Returned Condition Codes

When a function returns a condition code to your program unit, you should always examine the returned condition code. To check for a failure condition (warning, error, or severe error), test the returned condition code for a logical value of false. The following program segment invokes the run-time library procedure LIB$DATE_TIME, checks the returned condition code (returned in the variable STATUS), and, if an error has occurred, signals the condition code by calling the run-time library procedure LIB$SIGNAL (Section 9.1.3 describes signaling):

```
INTEGER*4 STATUS,
2         LIB$DATE_TIME
CHARACTER*23 DATE

STATUS = LIB$DATE_TIME (DATE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

To check for a specific error, test the return status for a particular condition code. For example, LIB$DATE_TIME returns a success code (LIB$_STRTRU) when it truncates the string. If you want to take special action when truncation occurs, specify the condition as shown in the following example (the special action would follow the IF statement):

```
INTEGER*4 STATUS,
2         LIB$DATE_TIME
CHARACTER*23 DATE

INCLUDE '($LIBDEF)'
    .
    .
    .
STATUS = LIB$DATE_TIME (DATE)
IF (STATUS .EQ. LIB$_STRTRU) THEN
    .
    .
    .
```

9.1.2.2 Testing SS$_NOPRIV and SS$_EXQUOTA

The SS$_NOPRIV and SS$_EXQUOTA condition codes returned by a number of system service procedures require special checking. Any system service that is listed as returning SS$_NOPRIV or SS$_EXQUOTA can, instead, return a more specific condition code that indicates the privilege or quota in question. Table 9–1 list the specific privilege errors and Table 9–2 lists the quota errors.

Table 9–1 Privilege Errors

SS$_NOACNT	SS$_NOALLSPOOL	SS$_NOALTPRI
SS$_NOBUGCHK	SS$_NOBYPASS	SS$_NOCMEXEC
SS$_NOCMKRNL	SS$_NODETACH	SS$_NODIAGNOSE
SS$_NODOWNGRADE	SS$_NOEXQUOTA	SS$_NOGROUP
SS$_NOGRPNAM	SS$_NOGRPPRV	SS$_NOLOGIO
SS$_NOMOUNT	SS$_NONETMBX	SS$_NOOPER

Table 9–1 (Cont.) Privilege Errors

SS$_NOPFNMAP	SS$_NOPHYIO	SS$_NOPRMCEB
SS$_NOPRMGBL	SS$_NOPRMMBX	SS$_NOPSWAPM
SS$_NOREADALL	SS$_NOSECURITY	SS$_NOSETPRV
SS$_NOSHARE	SS$_NOSHMEM	SS$_NOSYSGBL
SS$_NOSYSLCK	SS$_NOSYSNAM	SS$_NOSYSPRV
SS$_NOTMPMBX	SS$_NOUPGRADE	SS$_NOVOLPRO
SS$_NOWORLD		

Table 9–2 Quota Errors

SS$_EXASTLM	SS$_EXBIOLM	SS$_EXBYTLM
SS$_EXDIOLM	SS$_EXENQLM	SS$_EXFILLM
SS$_EXPGFLQUOTA	SS$_EXPRCLM	SS$_EXTQELM

Since either a general or a specific code can be returned, your program must test for both. The following four symbols provide a starting and ending point with which you can compare the returned condition code:

- SS$_NOPRIVSTRT—First specific code for SS$_NOPRIV
- SS$_NOPRIVEND—Last specific code for SS$_NOPRIV
- SS$_NOQUOTASTRT—First specific code for SS$_EXQUOTA
- SS$_NOQUOTAEND—Last specific code for SS$_EXQUOTA

The following VAX FORTRAN example tests for a privilege error by comparing STATUS (the returned condition code) with the specific condition code SS$_NOPRIV and the range provided by SS$_NOPRIVSTRT and SS$_NOPRIVEND. You would test for SS$_NOEXQUOTA in a similar fashion.

```
   .
   .
   .
! Declare status and status values
INTEGER STATUS
INCLUDE '($SSDEF)'
   .
   .
   .
IF (.NOT. STATUS) THEN
  IF ((STATUS .EQ. SS$_NOPRIV) .OR.
2     ((STATUS .GE. SS$_NOPRIVSTRT) .AND.
2      (STATUS .LE. SS$_NOPRIVEND))) THEN
   .
   .
   .
  ELSE
   CALL LIB$SIGNAL (%VAL(STATUS))
  END IF
END IF
```

9.1.3 Signaling Mechanism

Signaling a condition code causes the VMS operating system to pass control to a special subprogram called a condition handler. The VMS operating system invokes a default condition handler unless you have established your own. The default condition handler displays the associated error message and continues or, if the error is a severe error, terminates program execution (see Section 9.1.3.1).

You can signal a condition code by invoking the run-time library procedure LIB$SIGNAL and passing the condition code as the first argument. (The *VMS Run-Time Library Routines Volume* contains the complete specifications for LIB$SIGNAL.) The following statement signals the condition code contained in the variable STATUS.

```
CALL LIB$SIGNAL (%VAL(STATUS))
```

When an error occurs in a subprogram, the subprogram can signal the appropriate condition code rather than return the condition code to the invoking program unit. In addition, some statements also signal condition codes; for example, an assignment statement that attempts to divide by zero signals the condition code SS$_INTDIV.

9.1.3.1 Default Condition Handling

VMS has two default condition handlers: the traceback and catchall handlers. The traceback handler is in effect if you link your program with the /TRACEBACK qualifier of the LINK command (the default). Once you have completed program development, you generally link your program with the /NOTRACEBACK qualifier and use the catchall handler.

- Traceback handler—Displays the message associated with the signaled condition code, the traceback message, the program unit name and line number of the statement that signaled the condition code, and the relative and absolute program counter values. (On a warning or error, the number of the next statement to be executed is displayed.) In addition, the traceback handler displays the names of the program units in the calling hierarchy and the line numbers of the invocation statements. After displaying the error information, the traceback handler continues program execution or, if the error is severe, terminates program execution.

- Catchall handler—Displays the message associated with the condition code and then continues program execution or, if the error is severe, terminates execution. The catchall handler is not invoked if the traceback handler is enabled.

For example, if the condition code INCOME_LINELOST is signaled at line 496 of GET_STATS, regardless of which default handler is in effect, the following message is displayed:

```
%INCOME-W-LINELOST, Statistics on last line lost due to CTRL/Z
```

If the traceback handler is in effect, the following text is also displayed:

```
%TRACE-W-TRACEBACK, symbolic stack dump follows
module name      routine name        line       rel PC    abs PC

GET_STATS        GET_STATS            497      00000306  00008DA2
INCOME           INCOME               148      0000015A  0000875A
                                               0000A5BC  0000A5BC
                                               00009BDB  00009BDB
                                               0000A599  0000A599
```

Because INCOME_LINELOST is a warning, the line number of the next statement to be executed (497), rather than the line number of the statement that signaled the condition code, is displayed. Line 148 of the program unit INCOME invoked GET_STATS.

9.1.3.2 Changing a Signal to a Return Status

If you expect a particular condition code to be signaled, you can prevent the VMS operating system from invoking the default condition handler by establishing a different condition handler. The following paragraphs describe how to establish and use the system-defined condition handler LIB$SIG_TO_RET, which changes a signal to a return status that your program can examine. For more information on condition handlers, see Section 9.3.

To change a signal to a return status, you must put any code that might signal a condition code into a function where the function value is a return status. The function containing the code must perform the following operations:

- Declare LIB$SIG_TO_RET—Declare the condition handler LIB$SIG_TO_RET.
- Establish LIB$SIG_TO_RET—Invoke the run-time library procedure LIB$ESTABLISH to establish a condition handler for the current program unit. Specify the name of the condition handler LIB$SIG_TO_RET as the only argument.
- Initialize the function value—Initialize the function value to SS$_NORMAL so that, if no condition code is signaled, the function returns a success status to the invoking program unit.
- Declare necessary dummy arguments—If any statement that might signal a condition code is a subprogram that requires dummy arguments, pass the necessary arguments to the function. In the function, declare each dummy argument exactly as it is declared in the subprogram that requires it and specify the dummy arguments in the subprogram invocation.

If the program unit GET_1_STAT in the following function signals a condition code, LIB$SIG_TO_RET changes the signal to the return status of the INTERCEPT_SIGNAL function and returns control to the program unit that invoked INTERCEPT_SIGNAL. (If GET_1_STAT has a condition handler established, VMS invokes that handler before invoking LIB$SIG_TO_RET.)

```
FUNCTION INTERCEPT_SIGNAL (STAT,
2                          ROW,
2                          COLUMN)

! Dummy arguments for GET_1_STAT
INTEGER STAT,
2       ROW,
2       COLUMN
! Declare SS$_NORMAL
INCLUDE '($SSDEF)'
! Declare condition handler
EXTERNAL LIB$SIG_TO_RET
```

```
! Declare user routine
INTEGER GET_1_STAT
! Establish LIB$SIG_TO_RET
CALL LIB$ESTABLISH (LIB$SIG_TO_RET)
! Set return status to success
INTERCEPT_SIGNAL = SS$_NORMAL
! Statements and/or subprograms that
! signal expected error condition codes
STAT = GET_1_STAT (ROW,
2                  COLUMN)

END
```

When the program unit that invoked INTERCEPT_SIGNAL regains control, it should check the return status (as shown in Section 9.1.2) to determine which condition code, if any, was signaled during execution of INTERCEPT_SIGNAL.

9.2 Defining Condition Codes and Messages

You can supplement system condition codes and messages by defining your own. To define your own condition codes and messages, follow these steps:

1. Create a message source file
2. Compile the message source file with the MESSAGE command
3. Link the resultant object module with your program

9.2.1 Creating the Message Source File

A message source file contains definition statements and directives. The following source message file defines the error messages generated by the example INCOME program:

INCMSG.MSG

```
.FACILITY INCOME, 1 /PREFIX=INCOME__

.SEVERITY WARNING
    LINELOST   "Statistics on last line lost due to CTRL/Z"

.SEVERITY SEVERE
    BADFIXVAL  "Bad value on /FIX"
    CTRLZ      "CTRL/Z entered on terminal"
    FORIOERR   "FORTRAN I/O error"
    INSFIXVAL  "Insufficient values on /FIX"
    MAXSTATS   "Maximum number of statistics already entered"
    NOACTION   "No action qualifier specified"
    NOHOUSE    "No such house number"
    NOSTATS    "No statistics to report"

.END
```

The default file type of a message source file is MSG. For a complete description of the MESSAGE Utility, see the *VMS Message Utility Manual*.

9.2.1.1 Specifying the Facility

To specify the name and number of the facility for which you are defining the error messages, use the .FACILITY directive. For instance, the following .FACILITY directive specifies the facility (program) INCOME and a facility number of 1:

```
.FACILITY INCOME, 1
```

In addition to identifying the program associated with the error messages, the .FACILITY directive specifies the facility prefix that is added to each condition name to create the symbolic name used to reference the message. By default, the prefix is the facility name followed by an underscore. For example, a condition name BADFIXVAL defined following the previous .FACILITY directive is referenced as INCOME_BADFIXVAL. You can specify a prefix other than the specified program name by specifying the /PREFIX qualifier of the .FACILITY directive.

By convention, system-defined condition codes are identified by the facility name, followed by a dollar sign, an underscore, and the condition name. User-defined condition codes are identified by the facility name, followed by two underscores, and the condition name. To include two underscores in the symbolic name, use the /PREFIX qualifier to specify the prefix.

```
.FACILITY INCOME, 1 /PREFIX=INCOME__
```

A condition name BADFIXVAL defined following this .FACILITY directive is referenced as INCOME__BADFIXVAL.

The facility number, which must be between 1 and 2047, is part of the condition code that identifies the error message. To prevent different programs from generating the same condition codes, the facility number must be unique. A good way to ensure uniqueness is to have the system manager keep a list of programs and their facility numbers in a file.

All messages defined after a .FACILITY directive are associated with the specified program. Specify either an .END directive or another .FACILITY directive to end the list of messages for that program. It is recommended that you have one .FACILITY directive per message file.

9.2.1.2 Specifying the Severity

Use the .SEVERITY directive and one of the following keywords to specify the severity of one or more condition codes:

SUCCESS
INFORMATIONAL
WARNING
ERROR
SEVERE or FATAL

All condition codes defined after a .SEVERITY directive have the specified severity (unless you use the /SEVERITY qualifier of the message definition statement to change the severity of one particular condition code). Specify an .END directive or another .SEVERITY directive to end the group of errors with the specified severity. Note that when the .END directive is used to end the list of messages for a .SEVERITY directive, it also ends the list of messages for the previous .FACILITY directive. The following example defines one condition code with a severity of WARNING and two condition codes with a severity of SEVERE. The optional spacing between the lines and at the beginning of the lines is used for clarity.

```
.SEVERITY WARNING
    LINELOST  "Statistics on last line lost due to CTRL/Z"

.SEVERITY SEVERE
    BADFIXVAL "Bad value on /FIX"
    INSFIXVAL "Insufficient values on /FIX"

.END
```

9.2.1.3 Specifying Condition Names and Messages

To define a condition code and message, specify the condition name and the message text. The condition name, when combined with the facility prefix, can contain up to 31 characters. The message text can be up to 255 characters but only 1 line long. Use quotation marks (" ") or angle brackets (<>) to enclose the text of the message. For example, the following line from INCMSG.MSG defines the condition code INCOME__BADFIXVAL:

```
BADFIXVAL  "Bad value on /FIX"
```

9.2.1.4 Specifying Variables in the Message Text

To include variables in the message text, specify formatted ASCII output (FAO) directives (for details, see the description of the Message utility in the *VMS Message Utility Manual*). Specify the /FAO_COUNT qualifier after either the condition name or the message text to indicate the number of FAO directives that you used. The following example includes an integer variable in the message text:

```
NONUMBER <No such house number:  !UL.  Try again.>/FAO_COUNT=1
```

The FAO directive !UL converts a longword to decimal notation. To include a character string variable in the message, you could use the FAO directive !AS, as shown in the following example:

```
NOFILE <No such file:  !AS.  Try again.>/FAO_COUNT=1
```

If the message text contains FAO directives, you must specify the appropriate variables when you signal the condition code (see Section 9.2.3).

9.2.2 Compiling and Linking the Messages

Use the DCL command MESSAGE to compile a message source file into an object module. The following command compiles the message source file INCMSG.MSG into an object module named INCMSG in the file INCMSG.OBJ:

```
$ MESSAGE INCMSG
```

To specify an object file name different from the source file name, use the /OBJECT qualifier of the MESSAGE command. To specify an object module name different from the source file name, use the .TITLE directive in the message source file.

9.2.2.1 Linking the Message Object Module

The message object module must be linked with your program so that the system can reference the messages. To simplify linking a program with the message object module, include the message object module in the program's object library. For example, to include the message module in INCOME's object library, enter the following:

```
$ LIBRARY INCOME.OLB INCMSG.OBJ
```

9.2.2.2 Accessing the Message Object Module from Multiple Programs

Including the message module in the program's object library does not allow other programs access to the module's condition codes and messages. To allow several programs access to a message module, create a default message library as follows:

1. Create the message library—Create an object module library and enter all of the message object modules into it.
2. Make the message library a default library—Equate the complete file specification of the object module library with the logical name LNK$LIBRARY. (If LNK$LIBRARY is already assigned a library name, you can create LNK$LIBRARY_1, LNK$LIBRARY_2, and so on.) By default, the linker searches any libraries equated with the LNK$LIBRARY logical names.

The following example creates the message library MESSAGLIB.OLB, enters the message object module INCMSG.OBJ into it, and makes MESSAGLIB.OLB a default library:

```
$ LIBRARY/CREATE MESSAGLIB
$ LIBRARY/INSERT MESSAGLIB INCMSG
$ DEFINE LNK$LIBRARY SYS$DISK:MESSAGLIB
```

9.2.2.3 Modifying a Message Source Module

To modify a message in the message library, modify and recompile the message source file, and then replace the module in the object module library. To access the modified messages, a program must relink against the object module library (or the message object module). The following command enters the module INCMSG into the message library MESSAGLIB; if MESSAGLIB already contains an INCMSG module, it is replaced:

```
$ LIBRARY/REPLACE MESSAGLIB INCMSG
```

9.2.2.4 Accessing Modified Messages Without Relinking

To allow a program to access modified messages without relinking, create a message pointer file. Message pointer files are useful if you need to provide messages in more than one language or frequently change the text of existing messages. See the description of the Message Utility in the *VMS Message Utility Manual.*

9.2.3 Signaling User-Defined Codes and Messages

To signal a user-defined condition code, you use the symbol formed by the facility prefix and the condition name (for example, INCOME__BADFIXVAL). Typically, you reference a condition code as a global symbol; however, you can create an include file (similar to the modules in the system library SYS$LIBRARY:FORSTSDEF.TLB) to define the condition codes as local symbols. If the message text contains FAO arguments, you must specify parameters for those arguments when you signal the condition code.

9.2.3.1 Signaling with Global Symbols

To signal a user-defined condition code using a global symbol, declare the appropriate condition code in the appropriate section of the program unit, then invoke the RTL routine LIB$SIGNAL to signal the condition code. The following statements signal the condition code INCOME__NOHOUSE when the value of FIX_HOUSE_NO is less than 1 or greater than the value of TOTAL_HOUSES:

```
EXTERNAL INCOME__NOHOUSE
    .
    .
    .
IF ((FIX_HOUSE_NO .GT. TOTAL_HOUSES) .OR.
2    FIX_HOUSE_NO .LT. 1)) THEN
  CALL LIB$SIGNAL (%VAL (%LOC (INCOME__NOHOUSE)))
  END IF
```

9.2.3.2 Signaling with Local Symbols

To signal a user-defined condition code using a local symbol, you must first create a file containing PARAMETER statements that equate each condition code with its value. To create such a file, do the following:

1. Create a listing file—Compile the message source file with the /LIST qualifier of the MESSAGE command. The /LIST qualifier produces a listing file with the same name as the source file and a file type of LIS. The following line might appear in a listing file:

   ```
   08018020      11 NOHOUSE   "No such house number"
   ```

   The hexadecimal value in the left-hand column is the value of the condition code; the decimal number in the second column is the line number; the text in the third column is the condition name; and the text in quotation marks is the message text.

2. Edit the listing file—For each condition name, define the matching condition code as a longword variable and use a language statement to equate the condition code to its hexadecimal condition value.

   Assuming a prefix of INCOME__, editing the previous statement would result in the following statements:

   ```
   INTEGER INCOME__NOHOUSE
   PARAMETER (INCOME__NOHOUSE = '08018020'X)
   ```

3. Rename the listing file—Name the edited listing file using the same name as the source file and a file type for your programming language (for example, FOR for VAX FORTRAN).

In the definition section of your program unit, declare the local symbol definitions by naming your edited listing file in an INCLUDE statement. (You must still link the message object file with your program.) Invoke the RTL routine LIB$SIGNAL to signal the condition code. The following statements signal the condition code INCOME__NOHOUSE when the value of FIX_HOUSE_NO is less than 1 or greater than the value of TOTAL_HOUSES:

```
! Specify the full file specification
INCLUDE '$DISK1:[DEV.INCOME]INCMSG.FOR'
    .
    .
    .
IF ((FIX_HOUSE_NO .GT. TOTAL_HOUSES) .OR.
2    FIX_HOUSE_NO .LT. 1)) THEN
  CALL LIB$SIGNAL (%VAL (INCOME__NOHOUSE))
END IF
```

9.2.3.3 Specifying FAO Parameters

If the message contains FAO arguments, you must specify the number of FAO arguments as the second argument of LIB$SIGNAL, the first FAO argument as the third argument, the second FAO argument as the fourth argument, and so on. Pass string FAO arguments by descriptor (the default). For example, to signal the condition code INCOME__NONUMBER, where FIX_HOUSE_NO contains the erroneous house number, specify the following:

```
EXTERNAL INCOME__NONUMBER
    .
    .
    .
IF ((FIX_HOUSE_NO .GT. TOTAL_HOUSES) .OR.
2    FIX_HOUSE_NO .LT. 1)) THEN
  CALL LIB$SIGNAL (%VAL (%LOC (INCOME__NONUMBER)),
2                %VAL (1),
2                %VAL (FIX_HOUSE_NO))
  END IF
```

To signal the condition code NOFILE, where FILE_NAME contains the invalid file specification, specify the following:

```
EXTERNAL INCOME__NOFILE
    .
    .
    .
IF (IOSTAT .EQ. FOR$IOS_FILNOTFOU)
2  CALL LIB$SIGNAL (%VAL (%LOC (INCOME__NOFILE)),
2                  %VAL (1),
2                  FILE_NAME)
```

Both of the previous examples use global symbols for the condition codes. You could use local symbols, as described in Section 9.2.3.2.

9.3 Condition Handlers

When a program signals a condition code, the VMS operating system searches for a condition handler, invokes the first handler it finds, and passes the information to the handler about the condition code and the state of the program when the condition code was signaled. If the handler resignals, the VMS operating system searches for another handler; otherwise, the search for a condition handler ends.

The VMS operating system searches for condition handlers in the following sequence:

1 Primary exception vectors—Four vectors (lists) of one or more condition handlers; each vector is associated with an access mode. By default, all of the primary exception vectors are empty. Exception vectors are primarily used for system programming, not application programming.

The debugger uses the primary exception vector associated with user mode.

When an exception occurs, the VMS operating system searches the primary exception associated with the access mode at which the exception occurred. To enter or cancel a condition handler in an exception vector, use the SYS$SETEXV system service. Condition handlers entered into the exception vectors associated with kernel, executive, and supervisor modes remain in effect until they are cancelled or until you log out. Condition handlers entered into the exception vector associated with user mode remain in effect until they are canceled or the image that entered them exits.

2 Secondary exception vectors—A set of exception vectors with the same structure as the primary exception vectors. Exception vectors are primarily used for system programming, not application programming. By default, all of the secondary exception vectors are empty.

3 Call frame condition handlers—Each program unit can establish one condition handler (the address of the handler is placed in the call frame of the program unit). The VMS operating system searches for condition handlers established by your program, beginning with the current program unit. If the current program unit has not established a condition handler, the VMS operating system searches for a handler established by the program unit that invoked the current program unit and so on back to the main program.

4 Traceback handler—If you do not establish any condition handlers and link your program with the /TRACEBACK qualifier of the LINK command (the default), the VMS operating system finds and invokes the traceback handler (see Section 9.1.3.1).

5 Catchall handler—If you do not establish any condition handlers and link your program with the /NOTRACEBACK qualifier of the LINK command, VMS finds and invokes the catchall handler (see Section 9.1.3.1).

6 Last-chance exception vectors—A set of exception vectors with the same structure as the primary and secondary exception vectors. Exception vectors are primarily used for system programming, not application programming. By default, the user- and supervisor-mode last-chance exception vectors are empty. The executive- and kernel-mode last-chance exception vectors contain procedures that cause a bugcheck (a nonfatal bugcheck results in an error log entry; a fatal bugcheck results in a system shutdown). The debugger uses the user-mode last-chance exception vector and DCL uses the supervisor-mode last-chance exception vector.

In cases where the default condition handling is insufficient, you can use the RTL routine LIB$ESTABLISH to establish your own handler. Typically, you need condition handlers only if your program must perform one of the following operations:

- Respond to condition codes that are signaled rather than returned, such as an integer overflow error. (Section 9.1.3.2 describes the system-defined handler LIB$SIG_TO_RET that allows you to treat signals as return values; Section 9.3.5 describes other useful system-defined handlers for arithmetic errors.)

- Modify part of a condition code, such as the severity (see Section 9.3.4 for more information). If you want to change the severity of any condition code to a severe error, you can use the run-time library procedure LIB$STOP instead of writing your own condition handler.
- Add additional messages to the one associated with the originally signaled condition code or log the messages associated with the originally signaled condition code (see Section 9.3.4 for more information).

9.3.1 Establishing a Condition Handler

To establish a condition handler for the current program unit, use the run-time library procedure LIB$ESTABLISH. The following program segment establishes the condition handler ERRLOG. Since the condition handler is used as an actual argument, it must be declared in an EXTERNAL statement.

```
INTEGER*4 OLD_HANDLER
EXTERNAL  ERRLOG
    .
    .
    .
OLD_HANDLER = LIB$ESTABLISH (ERRLOG)
```

As its function value, LIB$ESTABLISH returns the address of the previous handler. If only part of a program unit requires a special condition handler, you can reestablish the original handler by invoking LIB$ESTABLISH and specifying the saved handler address as follows:

```
CALL LIB$ESTABLISH (OLD_HANDLER)
```

9.3.2 Writing a Condition Handler

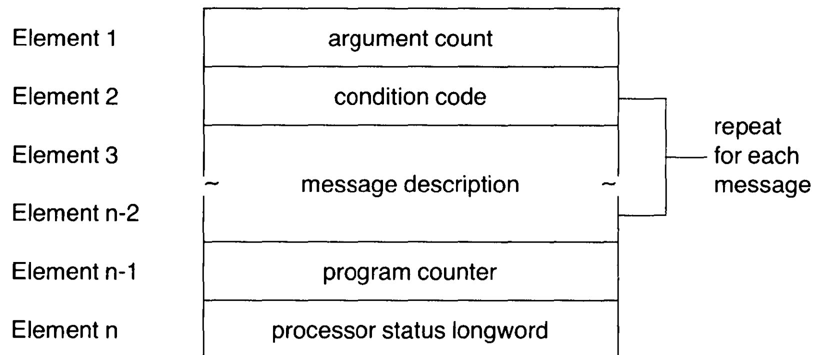
The VMS operating system passes two arrays to a condition handler. Any condition handler that you write should declare two dummy arguments as variable-length arrays, as in the following:

```
INTEGER*4 FUNCTION HANDLER (SIGARGS,
2                            MECHARGS)

INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
    .
    .
    .
```

9.3.2.1 The Signal Array

The first dummy argument, the signal array, describes the signaled condition codes that indicate which error occurred and the state of the process when the condition code was signaled. Figure 9–2 illustrates the the structure of a signal array.

Figure 9–2 Structure of a Signal Array



- Argument count—The number of elements in the array, not counting this first element.
- Condition code—The value of the condition code. If more than one message is associated with the error, this is the condition code of the first message.
- Message description—The format of the message description varies depending on the type of message being signaled. For more information, see the SYS$PUTMSG description in the *VMS System Services Reference Manual.*
- Program counter (PC)—If the error that caused the signal was a fault (occurring during the instruction's execution), the PC contains the address of the instruction that signaled the condition code. If the error that caused the signal was a trap (occurring at the end of the instruction), the PC contains the address of the instruction following the one that signaled the condition code. The error generated by LIB$SIGNAL is a trap.
- Processor status longword (PSL)—The PSL describes the state of the program at the time of the signal.

Typically, a condition handler does not use the PC or PSL.

9.3.2.2 The Mechanism Array

The second dummy argument, the mechanism array, describes the state of the process when the condition code was signaled. Typically, a condition handler references only the call depth and the saved function value. Currently, the mechanism array contains exactly five elements; however, since its length is subject to change, you should declare the dummy argument as a variable-length array. Figure 9–3 illustrates the structure of a mechanism array.

Figure 9–3 Structure of a Mechanism Array

Element 1	argument count
Element 2	establisher
Element 3	call depth
Element 4	function value
Element 5	R1

ZK-2051-84

- Argument count—The number of elements in the array not counting this first element (that is, four).
- Establisher—Pointer to information that allows the VMS operating system to resume execution of the program unit that established the condition handler.
- Call depth—The number of program units called between the program unit that established the handler and the program unit that signaled the condition code. For example, if a program unit establishes a handler and then signals a condition code, the call depth is 0. If a program unit establishes a handler and then calls a subprogram that signals a condition code, the call depth is 1, and so on.
- R0 and R1—The contents of the R0 and R1 registers.

A condition handler is usually written in anticipation of a particular set of condition codes. Since a handler is invoked in response to any signaled condition code, you should begin your handler by comparing the condition code passed to the handler (element 2 of the signal array) against the condition codes expected by the handler. If the signaled condition code is not one of the expected codes, you should resignal the condition code by equating the function value of the handler to the global symbol SS$_RESIGNAL.

9.3.2.3 Comparing the Signaled Condition with an Expected Condition

To compare the signaled condition code to a list of expected condition codes, use the RTL routine LIB$MATCH_COND. The first argument passed to LIB$MATCH_COND is the signaled condition code, the second element of the signal array. The rest of the arguments passed to LIB$MATCH_COND are the expected condition codes. LIB$MATCH_COND compares the first argument with each of the remaining arguments and returns the number of the argument that matches the first one. For example, if the second argument matches the first argument, LIB$MATCH_COND returns a value of 1. If the first argument does not match any of the other arguments, LIB$MATCH_COND returns 0.

The following condition handler determines whether the signaled condition code is one of four VAX FORTRAN I/O errors. If it is not, the condition handler resignals the condition code. Note that, when a VAX FORTRAN I/O error is signaled, the signal array describes VMS condition code, not the VAX FORTRAN error code.

```
INTEGER FUNCTION HANDLER (SIGARGS,
2                         MECHARGS)

! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
INCLUDE '($FORDEF)'   ! Declare the FOR$_ symbols
INCLUDE '($SSDEF)'    ! Declare the SS$_ symbols
INTEGER INDEX
! Declare procedures
INTEGER LIB$MATCH_COND
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       FOR$_FILNOTFOU,
2                       FOR$_OPEFAI,
2                       FOR$_NO_SUCDEV,
2                       FOR$_FILNAMSPE)
IF (INDEX .EQ. 0) THEN
  ! Not an expected condition code, resignal
  HANDLER = SS$_RESIGNAL
ELSE IF (INDEX .GT. 0) THEN
  ! Expected condition code, handle it
    .
    .
    .
END IF

END
```

9.3.2.4 Exiting From the Condition Handler

You can exit from a condition handler in one of three ways:

- Continue execution of the program—If you equate the function value of the condition handler to SS$_CONTINUE, the condition handler returns control to the program at the statement that signaled the condition (fault) or the statement following the one that signaled the condition (trap). The RTL routine LIB$SIGNAL generates a trap so that control is returned to the statement following the call to LIB$SIGNAL.

  In the following example, if the condition code is one of the expected codes, the condition handler displays a message (Section 9.3.4.2 describes how to display a message) and then returns the value SS$_CONTINUE to resume program execution:

```
INTEGER FUNCTION HANDLER (SIGARGS,
2                         MECHARGS)

! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
INCLUDE '($FORDEF)'
INCLUDE '($SSDEF)'
INTEGER*4 INDEX,
2         LIB$MATCH_COND
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       FOR$_FILNOTFOU,
2                       FOR$_OPEFAI,
2                       FOR$_NO_SUCDEV,
2                       FOR$_FILNAMSPE)
```

```
IF (INDEX .GT. 0) THEN
   .
   .
   .
             ! Display the message
   .
   .
   .
  HANDLER = SS$_CONTINUE
END IF
```

- Resignal the condition code—If you equate the function value of the condition handler to SS$_RESIGNAL or do not specify a function value (function value of 0), the handler allows the VMS operating system to execute the next condition handler. If you modify the signal array or mechanism array before resignaling, the modified arrays are passed to the next condition handler.

  In the following example, if the condition code is not one of the expected codes, the handler resignals:

```
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                        FOR$_FILNOTFOU,
2                        FOR$_OPEFAI,
2                        FOR$_NO_SUCDEV,
2                        FOR$_FILNAMSPE)

IF (INDEX .EQ. 0) THEN
  HANDLER = SS$_RESIGNAL
END IF
```

- Continue execution of the program at a previous location—If you call the SYS$UNWIND system service, the condition handler can return control to any point in the program unit that incurred the exception, the program unit that invoked the program unit that incurred the exception, and so on back to the program unit that established the condition handler.

9.3.2.5 Returning Control to the Program

Since correctly invoking the SYS$UNWIND system service requires a knowledge of VMS internals beyond the scope of this manual, your handlers should return control either to the program unit that established the handler or the program unit that invoked the program unit that established the handler.

To return control to the program unit that established the handler, invoke SYS$UNWIND and pass the call depth (third element of the mechanism array) as the first argument with no second argument.

```
! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
   .
   .
   .
CALL SYS$UNWIND (MECHARGS(3),)
```

To return control to the caller of the program unit that established the handler, invoke SYS$UNWIND without passing any arguments.

```
! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
    .
    .
    .
CALL SYS$UNWIND (,)
```

The first argument of the SYS$UNWIND system service specifies the number of program units to unwind (remove from the stack). If you specify this argument at all, you should do so as shown in the previous example. (MECHARGS(3) contains the number of program units that must be unwound to reach the program unit that established the handler that invoked SYS$UNWIND.) The second argument of the SYS$UNWIND system service contains the location of the next statement to be executed. Typically, you omit the second argument to indicate that the program should resume execution at the statement following the last statement executed in the program unit that is regaining control.

Each time SYS$UNWIND removes a program unit from the stack it invokes the condition handler (if any) established by that program unit, passing the condition handler the SS$_UNWIND condition code. To prevent the condition handler from resignaling the SS$_UNWIND condition code (and so complicating the unwind operation), you should include SS$_UNWIND as an expected condition code when you invoke LIB$MATCH_COND. When the condition code is SS$_UNWIND, your condition handler may perform necessary cleanup operations or do nothing.

In the following example, if the condition code is SS$_UNWIND, no action is performed. If the condition code is another of the expected codes, the handler displays the message and then returns control to the program unit that called the program unit that established the condition handler.

```
INTEGER FUNCTION HANDLER (SIGARGS,
2                         MECHARGS)

! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
INCLUDE '($FORDEF)'
INCLUDE '($SSDEF)'
INTEGER*4 INDEX,
2         LIB$MATCH_COND
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       SS$_UNWIND,
2                       FOR$_FILNOTFOU,
2                       FOR$_OPEFAI,
2                       FOR$_NO_SUCDEV,
2                       FOR$_FILNAMSPE)
IF (INDEX .EQ. 0) THEN
  ! Unexpected condition, resignal
  HANDLER = SS$_RESIGNAL
ELSE IF (INDEX .EQ. 1) THEN
  ! Unwinding, do nothing
```

```
ELSE IF (INDEX .GT. 1) THEN
    .
    .
    .
              ! Display the message
    .
    .
    .
  CALL SYS$UNWIND (,)
END IF
```

9.3.3 Debugging

You can debug a condition handler as you would any subprogram, except that you cannot use the DEBUG command STEP/INTO to enter a condition handler. You must set a breakpoint in the handler and wait for the debugger to invoke the handler.

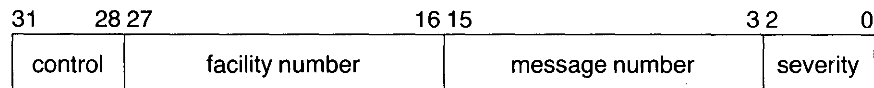
Typically, to trace execution of a condition handler, you set breakpoints at the statement in your program that should signal the condition code, at the statement following the one that should signal, and at the first executable statement in your condition handler.

9.3.4 Condition Handler Functions

The following sections describe some of the common functions performed by condition handlers. Since a condition handler cannot know exactly where you are in your program, you should avoid manipulating data or performing other mainline activities.

9.3.4.1 Modifying Condition Codes

As described in Figure 9–1, a condition code contains the following information:



To modify a condition code, copy a series of bits from one longword to another longword. For example, the following statement copies the first three bits (bits <2:0>) of STS$K_INFO to the first three bits of the signaled condition code, which is in the second element of the signal array named SIGARGS. As shown in the table in Section 9.1.1, STS$K_INFO contains the symbolic severity code for an informational message.

```
! Declare STS$K_ symbols
INCLUDE '($STSDEF)'
    .
    .
    .
! Change the severity of the condition code
! in SIGARGS(2) to informational
CALL MVBITS (STS$K_INFO,
2            0,
2            3,
2            SIGARGS(2),
2            0)
```

Once you modify the condition code, you can resignal the condition code and let the default condition handler display the associated message or use the SYS$PUTMSG system service to display the message. If your condition handler displays the message, do not resignal the condition code or the default condition handler will display the message a second time.

In the following example, the condition handler verifies that the signaled condition code is LIB$_NOSUCHSYM. If it is, the handler changes its severity from error to informational and then resignals the modified condition code. As a result of the handler's actions, the program displays an informational message indicating that the specified symbol does not exist and then continues executing.

```
INTEGER FUNCTION SYMBOL (SIGARGS,
2                        MECHARGS)
! Changes LIB$_NOSUCHSYM to an informational message

! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
! Declare index variable for LIB$MATCH_COND
INTEGER INDEX
! Declare condition codes
INCLUDE '($LIBDEF)'
INCLUDE '($STSDEF)'
INCLUDE '($SSDEF)'
! Declare library procedures
INTEGER LIB$MATCH_COND
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       LIB$NO_SUCHSYM)
! If the signaled condition code is LIB$NO_SUCHSYM,
! change its severity to informational.
IF (INDEX .GT. 0)
2  CALL MVBITS (STS$K_INFO,
2               0,
2               3,
2               SIGARGS(2),
2               0)

SYMBOL = SS$_RESIGNAL

END
```

9.3.4.2 Displaying Messages

The VMS operating system uses the SYS$PUTMSG system service to display messages. For consistency with the default handling mechanisms, you should use the same system service.

You can use the signal array that the VMS operating system passes to the condition handler as the first argument of the SYS$PUTMSG system service. The signal array contains the condition code, the number of required FAO arguments for each condition code, and the FAO arguments. The *VMS System Services Reference Manual* contains complete specifications for SYS$PUTMSG.

The last two array elements, the PC and PSL, are not FAO arguments and should be deleted before the array is passed to SYS$PUTMSG. Because the first element of the signal array contains the number of longwords in the array, you can effectively delete the last two elements of the array by subtracting 2 from the value in the first element. Before exiting from the condition handler, restore the last two elements of the array by adding 2 to the first element in case other handlers reference the array.

The following example performs the same function as the previous example. However, in this case, the condition handler uses the SYS$PUTMSG system service and then returns a value of SS$_CONTINUE so that the default handler is not executed.

```
INTEGER*4 FUNCTION SYMBOL (SIGARGS,
2                          MECHARGS)
    .
    .
    .
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       LIB$_NOSUCHSYM)
IF (INDEX .GT. 0) THEN
  ! If condition code is LIB$_NOSUCHSYM,
  ! change the severity to informational
  CALL MVBITS (STS$K_INFO,
2              0,
2              3,
2              SIGARGS(2),
2              0)

  ! Display the message
  SIGARGS(1) = SIGARGS(1) - 2   ! Subtract last two elements
  CALL SYS$PUTMSG (SIGARGS,,,)
  SIGARGS(1) = SIGARGS(1) + 2   ! Restore last two elements

  ! Continue program execution;
  SYMBOL = SS$_CONTINUE
ELSE
  ! Otherwise, resignal the condition
  SYMBOL = SS$_RESIGNAL
END IF

END
```

9.3.4.3 Chaining Messages

A condition handler can be used to add condition codes to an originally signaled condition code. For example, if your program calculates the standard deviation of a series of numbers and the user only enters one value, the VMS operating system signals the condition code SS$_INTDIV when the program attempts to divide by zero. (In calculating the standard deviation, the divisor is the number of values entered minus 1.) You could use a condition handler to add a user-defined message to the original message to indicate that only one value was entered.

To display multiple messages, pass the condition codes associated with the messages to the RTL routine LIB$SIGNAL. To display the message associated with an additional condition code, the handler must pass LIB$SIGNAL the condition code, the number of FAO arguments used, and the FAO arguments. To display the message associated with the originally signaled condition codes, the handler must pass LIB$SIGNAL each element of the signal array as a separate argument. Since the signal array is a variable-length array and LIB$SIGNAL cannot accept a variable number of arguments, when you write your handler, you must pass LIB$SIGNAL more arguments than you think will be required. Then, during execution of the handler, zero the arguments that you do not need (LIB$SIGNAL ignores zero values), as described in the following steps:

1. Declare an array with one element for each argument that you plan to pass LIB$SIGNAL. Fifteen elements are usually sufficient.

   ```
   INTEGER*4 NEWSIGARGS(15)
   ```

2. Transfer the condition codes and FAO information from the signal array to your new array. The first element and the last two elements of the signal array do not contain FAO information and should not be transferred.

3. Fill any remaining elements of your new array with zeros.

The following example demonstrates steps 2 and 3:

```
DO I = 1, 15

  IF (I .LE. SIGARGS(1) - 2) THEN
    NEWSIGARGS(I) = SIGARGS(I+1)   ! Start with SIGARGS(2)
    ELSE
    NEWSIGARGS(I) = 0              ! Pad with zeros
  END IF

END DO
```

Since the new array is a known-length array, you can specify each element as an argument to LIB$SIGNAL.

The following condition handler ensures that the signaled condition code is SS$_INTDIV. If it is, the user-defined message ONE_VALUE is added to SS$_INTDIV and both messages are displayed.

```
INTEGER FUNCTION HANDLER (SIGARGS,
2                         MECHARGS)

! Declare dummy arguments
INTEGER SIGARGS(*),
2       MECHARGS(*)
! Declare new array for SIGARGS
INTEGER NEWSIGARGS (15)
```

```
! Declare index variable for LIB$MATCH_COND
INTEGER INDEX
! Declare procedures
INTEGER LIB$MATCH_COND
! Declare condition codes
EXTERNAL ONE_VALUE
INCLUDE '($SSDEF)'
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       SS$_INTDIV)
IF (INDEX .GT. 0) THEN

  DO I=1,15
    IF (I .LE. SIGARGS(1) - 2) THEN
      NEWSIGARGS(I) = SIGARGS(I+1)  ! Start with SIGARGS(2)
    ELSE
      NEWSIGARGS(I) = 0             ! Pad with zeros
    END IF
  END DO

    ! Signal messages
  CALL LIB$SIGNAL (%VAL(NEWSIGARGS(1)),
2                  %VAL(NEWSIGARGS(2)),
2                  %VAL(NEWSIGARGS(3)),
2                  %VAL(NEWSIGARGS(4)),
2                  %VAL(NEWSIGARGS(5)),
2                  %VAL(NEWSIGARGS(6)),
2                  %VAL(NEWSIGARGS(7)),
2                  %VAL(NEWSIGARGS(8)),
2                  %VAL(NEWSIGARGS(9)),
2                  %VAL(NEWSIGARGS(10)),
2                  %VAL(NEWSIGARGS(11)),
2                  %VAL(NEWSIGARGS(12)),
2                  %VAL(NEWSIGARGS(13)),
2                  %VAL(NEWSIGARGS(14)),
2                  %VAL(NEWSIGARGS(15)),
2                  %VAL(%LOC(ONE_VALUE)),
2                  %VAL(0))

  HANDLER = SS$_CONTINUE
ELSE
  HANDLER = SS$_RESIGNAL

END IF

END
```

9.3.4.4 Logging Messages

When a program executes interactively or from within a command procedure, the logical names SYS$OUTPUT and SYS$ERROR are both equated to the user's terminal by default.

SYS$ERROR and SYS$OUTPUT

To write the error messages displayed by your program to a file as well as to the terminal, equate SYS$ERROR to a file specification. (When a program executes as a batch job, the logical names SYS$OUTPUT and SYS$ERROR are both equated to the batch log by default. To write error messages to the log file and a second file, equate SYS$ERROR to the second file.) Success messages are not written to SYS$ERROR.

Creating a Running Log of Messages Using SYS$PUTMSG

To keep a running log (that is, a log that is resumed each time your program is is invoked) of the messages displayed by your program, use SYS$PUTMSG. Create a condition handler that invokes SYS$PUTMSG regardless of the signaled condition code. When you invoke SYS$PUTMSG, specify a function that writes the formatted message to your log file and then returns with a function value of 0. Have the condition handler resignal the condition code. One of the arguments of the SYS$PUTMSG system service allows you to specify a user-defined function that SYS$PUTMSG invokes after formatting the message and before displaying the message. SYS$PUTMSG passes the specified function the formatted message. If the function returns with a function value of 0, SYS$PUTMSG does not display the message; if the function returns with a value of 1, SYS$PUTMSG displays the message. The *VMS System Services Reference Manual* contains complete specifications for SYS$PUTMSG.

Suppressing the Display of Messages in the Running Log

To keep a running log of messages, you might have your main program open a file for the error log, write the date, and then establish a condition handler to write all signaled messages to the error log. Each time a condition is signaled, a condition handler, like the one in the following example, would invoke SYS$PUTMSG and specify a function that writes the message to the log file and returns with a function value of 0. SYS$PUTMSG writes the message to the log file, but does not display the message. After SYS$PUTMSG writes the message to the log file, the condition handler resignals to continue program execution. (The condition handler uses LIB$GET_COMMON to read the unit number of the file from the per-process common block.)

ERR.FOR

```
INTEGER FUNCTION ERRLOG (SIGARGS,
2                        MECHARGS)
! Writes the message to file opened on the
! logical unit named in the per-process common block
! Define the dummy arguments
INTEGER SIGARGS(*),
2       MECHARGS(*)
INCLUDE '($SSDEF)'

EXTERNAL PUT_LINE
INTEGER PUT_LINE
! Pass signal array and PUT_LINE routine to SYS$PUTMSG
SIGARGS(1) = SIGARGS(1) - 2   ! Subtract PC/PSL from signal array
CALL SYS$PUTMSG (SIGARGS,
2               PUT_LINE, )
SIGARGS(1) = SIGARGS(1) + 2   ! Replace PC/PSL

ERRLOG = SS$_RESIGNAL

END
```

PUT_LINE.FOR

```
INTEGER FUNCTION PUT_LINE (LINE)
! Writes the formatted message in LINE to
! the file opened on the logical unit named
! in the per-process common block
! Dummy argument
CHARACTER*(*) LINE
! Logical unit number
CHARACTER*4 LOGICAL_UNIT
INTEGER UNIT_NUM
! Indicates that SYS$PUTMSG is not to display the message
PUT_LINE = 0
! Get logical unit number and change to integer
STATUS = LIB$GET_COMMON (LOGICAL_UNIT)
READ (UNIT = LOGICAL_UNIT,
2     FMT = '(I4)') UNIT_NUMBER
! The main program opens the error log
WRITE (UNIT = UNIT_NUMBER,
2      FMT = '(A)') LINE

END
```

9.3.5 System-Defined Arithmetic Condition Handlers

The VMS operating system provides the following arithmetic condition handlers:

- LIB$DEC_OVER—Enables or disables the signaling of a decimal overflow. By default, signaling is disabled.
- LIB$FLT_UNDER—Enables or disables the signaling of a floating-point underflow. By default, signaling is disabled.
- LIB$INT_OVER—Enables or disables the signaling of an integer overflow. By default, signaling is enabled.

You can establish these handlers in one of two ways:

- Invoke the appropriate handler as a function specifying the first argument as 1 to enable signaling.
- Invoke the handler with command qualifiers when you compile your program. Refer to your program language manuals.

9.4 Exit Handlers

When an image exits, the VMS operating system performs the following operations:

- Invokes any user-defined exit handlers.
- Invokes the system-defined default exit handler, which closes any files that were left open by the program or user-defined exit handlers.
- Executes a number of cleanup operations collectively known as image run-down. The following list contains some of these cleanup operations:
  - Cancels outstanding ASTs and timer requests.

- Deassigns any channel assigned by your program and not already deassigned by your program or the system.
- Deallocates devices allocated by the program.
- Disassociates common event flag clusters associated with the program.
- Deletes user-mode logical names created by the program (unless you specify otherwise, logical names created by SYS$CRELNM are user-mode logical names).
- Restores internal storage (for example, stacks or mapped sections) to its original state.

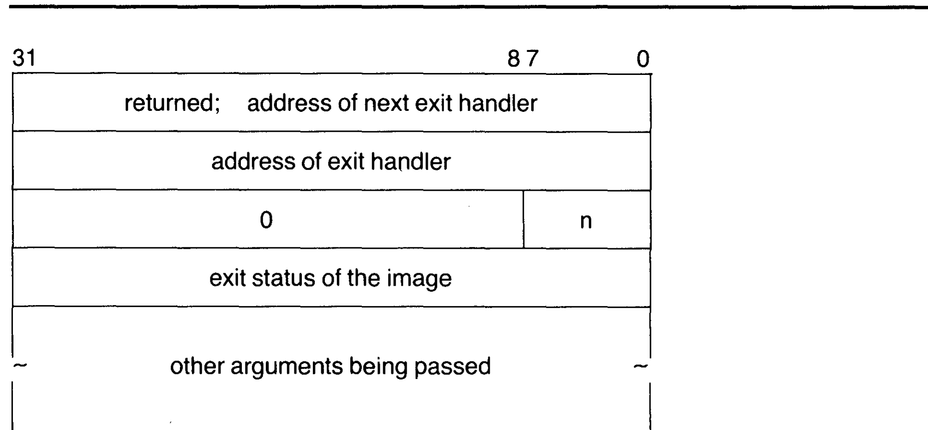
If any exit handler exits using the SYS$EXIT system service, none of the remaining handlers is executed. In addition, if an image is aborted by the DCL command STOP (the user presses CTRL/Y and then types STOP), the system performs image run-down and does not invoke any exit handlers. (The DCL command EXIT invokes the exit handlers before running down the image.)

Use exit handlers to perform any cleanup that your program requires in addition to the normal run-down operations performed by the VMS operating system. In particular, if your program must perform some final action regardless of whether it exits normally or is aborted, you should write and establish an exit handler to perform that action.

9.4.1 Establishing an Exit Handler

To establish an exit handler, use the SYS$DCLEXH system service. The SYS$DCLEXH system service requires one argument—a variable-length data structure that describes the exit handler. Figure 9–4 illustrates the structure of an exit handler.

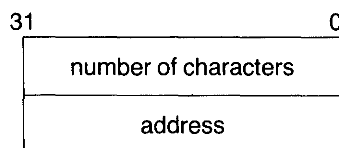
Figure 9–4 Structure of an Exit Handler



The first longword of the structure contains the address of the next handler. The VMS operating system uses this argument to keep track of the established exit handlers; do not modify this value. The second longword of the structure contains the address of the exit handler being established. The low-order byte of the third longword contains the number of arguments to be passed to the exit handler. Each of the remaining longwords contains the address of an argument.

The first argument passed to an exit handler is an integer value containing the final status of the exiting program. The status argument is mandatory. However, you should not supply the final status value; when the VMS operating system invokes an exit handler, it passes the handler the final status of the exiting program.

To pass an argument with a numeric data type, use programming language statements to assign the address of a numeric variable to one of the longwords in the exit handler data structure. To pass an argument with a character data type, create a descriptor of the following form:



Use the language statements to assign the address of the descriptor to one of the longwords in the exit handler data structure.

The following program segment establishes an exit handler with two arguments, the mandatory status argument and a character argument:

```
    .
    .
    .
! Arguments for exit handler
INTEGER EXIT_STATUS         ! Status
CHARACTER*12 STRING         ! String
STRUCTURE /DESCRIPTOR/
  INTEGER SIZE,
2         ADDRESS
END STRUCTURE
RECORD /DESCRIPTOR/ EXIT_STRING
! Setup for exit handler
STRUCTURE /EXIT_DESCRIPTOR/
 INTEGER LINK,
2        ADDR,
2        ARGS /2/,
2        STATUS_ADDR,
2        STRING_ADDR
END STRUCTURE
RECORD /EXIT_DESCRIPTOR/ HANDLER
! Exit handler
EXTERNAL EXIT_HANDLER
    .
    .
    .
! Set up descriptor
EXIT_STRING.SIZE = 12      ! Pass entire string
EXIT_STRING.ADDRESS = %LOC (STRING)
! Enter the handler and argument addresses
! into the exit handler description
HANDLER.ADDR = %LOC(EXIT_HANDLER)
HANDLER.STATUS_ADDR = %LOC(EXIT_STATUS)
HANDLER.STRING_ADDR = %LOC(EXIT_STRING)
! Establish the exit handler
CALL SYS$DCLEXH (HANDLER)
    .
    .
    .
```

An exit handler can be established at any time during your program and remains in effect until it is canceled (with SYS$CANEXH) or executed. If you establish more than one handler, the handlers are executed in reverse order: the handler established last is executed first; the handler established first is executed last.

9.4.2 Writing an Exit Handler

An exit handler should be written as a subroutine since no function value can be returned. The dummy arguments of the exit subroutine should agree in number, order, and data type with the arguments you specified in the call to SYS$DCLEXH.

Assume that two or more programs are cooperating with each other. To keep track of which programs are executing, each has been assigned a common event flag (the common event flag cluster is named ALIVE). When a program begins, it sets its flag; when the program terminates, it clears its flag. Since it is important that each program clear its flag before exiting, you create an exit handler (such as the one in the following example) to perform the action.

The exit handler accepts two arguments, the final status of the program and the number of the event flag to be cleared. Since, in this example, the cleanup operation is to be performed regardless of whether the program completes successfully, the final status is not examined in the exit routine. (This subroutine would not be used with the exit handler declaration in the previous example.)

CLEAR_FLAG.FOR

```
SUBROUTINE CLEAR_FLAG (EXIT_STATUS,
2                      FLAG)
! Exit handler clears the event flag

! Declare dummy argument
INTEGER EXIT_STATUS,
2       FLAG
! Declare status variable and system routine
INTEGER STATUS,
2       SYS$ASCEFC,
2       SYS$CLREF
! Associate with the common event flag
! cluster and clear the flag
STATUS = SYS$ASCEFC (%VAL(FLAG),
2                    'ALIVE',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

If for any reason you must perform terminal I/O from an exit handler, use appropriate RTL routines. Trying to access the terminal from an exit handler using language I/O statements may cause a redundant I/O error.

9.4.3 Debugging an Exit Handler

To debug an exit handler, you must set a breakpoint in the handler and wait for the VMS operating system to invoke that handler; you cannot use the DEBUG command STEP/INTO to enter an exit handler. In addition, when the debugger is invoked, it establishes an exit handler that exits using the SYS$EXIT system service. If you invoke the debugger when you invoke your image, the debugger's exit handler does not affect your program's handlers because the debugger's handler is established first and so executes last. However, if you invoke the debugger after your program begins executing (the user presses CTRL/Y and then types DEBUG), the debugger's handler may affect the execution of your program's exit handlers, since one or more of your handlers may have been established before the debugger's handler and so will not be executed.

10 Memory Management

Managing memory directly can improve program efficiency. The memory management routines allow you to allocate specific amounts of memory. Therefore, you need not be using more memory than required for your program.

You can use either RTL routines or system services to allocate and deallocate memory space. The RTL routines include routines for creating, deleting, and accessing information on virtual address space. You can either allocate a specified number of contiguous 512-byte pages or create a zone of virtual address space. A zone is a logical unit of memory that you can control as an independent area. It can be any size required by your program.

The system services also allocate and deallocate virtual address space. With system services, you can also control the process's working set size and map files into the process's virtual address space. System services provide more control over allocation procedures. However, you must also manage the allocation more precisely.

10.1 Using RTL Routines

The LIB$GET_VM_PAGE routine allows you to allocate a specified number of 512-byte pages for your program. The pages are allocated in user mode with read/write access, and they are contiguous. Because allocated pages are contiguous, you should not use LIB$GET_VM_PAGE to allocate a large number of pages (over 1000 pages) within a single request. For large requests, you should use system services.

For detailed information and examples using the memory management RTL routines, refer to the *VMS RTL Library (LIB$) Manual.*

The format for LIB$GET_VM_PAGE is as follows:

```
LIB$GET_VM_PAGE (num-pages, base-adr, [zone-id])
```

With this routine, you need to specify only the number of pages you need in the **num-pages** argument. The routine returns the base address of the contiguous block of pages that have been allocated in the **base-adr** argument.

Once you allocate pages with this routine, you must use LIB$FREE_VM_PAGE to deallocate the pages.

You can also use RTL routines to create zones of address space. A zone is a subdivision of address space that you can control as one unit. You might use a zone for storing short-lived data structures that you can subsequently delete all at once, for storing a program that does not reference a wide range of addresses, to specify a memory allocation algorithm specific to your program, or to specify attributes, like block size and alignment, specific to your program.

Use the LIB$CREATE_VM_ZONE routine to create a virtual address zone. The format for this routine is as follows:

LIB$CREATE_VM_ZONE (zone_id,[algorithm],[algorithm_arg] [,flags] [,extend_size],[initial_size],[block_size],[alignment],[page_limit],[p1])

For more information about LIB$CREATE_VM_ZONE, refer to the *VMS Run-Time Library Routines Volume*.

Allocating Address Space

Use the **algorithm** argument to specify how much space should be allocated—as a linked list of free blocks, as a set of lookaside lists indexes by request sizes, as a set of lookaside lists for some block sizes, or as a single queue of free blocks.

Allocating Pages Within the Zone

Use the **initial_size** argument to allocate a specified number of pages from the zone when it is created. Subsequent to zone creation, you can use LIB$GET_VM to allocate space.

Specifying the Block Size

Use the **block_size** argument to specify, in bytes, the block size.

Specifying Block Alignment

Use the **alignment** argument to specify, in bytes, the alignment for each block allocated.

Once a zone has been created and used, use LIB$DELETE_VM_ZONE to delete the zone and return the pages allocated to the process-wide page pool. LIB$RESET_VM_ZONE frees pages for subsequent allocation but does not delete the zone or return the pages to the process-wide page pool. Use LIB$SHOW_VM_ZONE to get information about a specific zone.

10.2 Using System Sevices

The system services provide extensive control over address space allocation by allowing you to do the following types of tasks:

- Add or delete virtual address space to the process's program (P0) or control (P1) regions.
- Add or delete virtual address space at a specific range of addresses.
- Increase or decrease the number of pages in a program's working set.
- Lock or delete pages of a program's working set in memory.
- Lock the entire program's working set in memory (by disabling process swapping).
- Map files to a particular section of memory.

For more information on mapping files, refer to Chapter 5. For detailed information and examples using the memory management system services, refer to the *VMS System Services Volume*.

10.2.1 Working with Address Space

The system services allow you to add address space anywhere within the process's program (P0) or control (P1) regions. To add address space at the end of the P0 or P1 region, use the SYS$EXPREG service. To add address space in other portions of the P0 or P1 regions, use SYS$CRETVA.

The format for SYS$EXPREG is as follows:

SYS$EXPREG (pagcnt,[retadr],[acmode],[region])

Specifying the Number of Pages

Use the **pagcnt** argument to specify the number of pages to add to the end of the region. The range of addresses where the new pages were added is returned in **retadr**.

Specifying the Access Mode

Use the **acmode** argument to specify the access to be assigned to the newly created pages.

Specifying the Region

Use the **region** argument to specify whether to add the pages to the end of the P0 or P1 region.

To deallocate pages allocated with SYS$EXPREG, use SYS$DELTVA.

To allocate address space at a specific area, you could use SYS$CRETVA. However, using SYS$CRETVA presents some risk because it can delete pages that already exist if those pages are not owned by a more privileged access mode. Further, if those pages are deleted, no notification is sent. Therefore, it is recommended that unless you have complete control over an entire system, you should use SYS$EXPREG or the RTL routines to allocate address space.

10.2.2 Adjusting Working Sets

The size of a working set for a program is set by default. To improve program efficiency, you may need to adjust the default value. If there is excess paging, you probably need to increase the working set size. If the program is small, you may not need the entire working set size allocated to your program.

SYS$ADJWSL allows you to either increase or decrease the working set size. The format for this routine is as follows:

SYS$ADJWSL ([pagcnt],[wsetlm])

Use the **pagcnt** argument to specify the number of pages to add or subtract from the current working set size. The new working set size is returned in **wsetlm**.

You can also lock a range of pages into the working set. Once locked into the working set, those pages remain until they are unlocked or program execution ends.

Use the SYS$LKWSET to lock specific pages in the working set. The format is as follows:

SYS$LKWSET (inadr,[retadr],[acmode])

Specifying a Range of Addresses

Use the **inadr** argument to specify the range of addresses to be locked. The range of addresses of the pages actually locked are returned in the **retadr** argument.

Specifying the Access Mode

Use the **acmode** argument to specify the access mode to be associated with the pages you want locked.

If you want to lock an entire process's pages into memory, use SYS$SETSWM. Also, you can use SYS$LCKPAG to lock specific pages in memory. These pages are not part of the process's working set, but they are forced into the process's working set. However, even if the process working set is swapped out, these pages remain in memory until they are unlocked with SYS$ULKPAG. To use either of these services, you need PSWAPM privilege.

To unlock pages in the working set, use SYS$ULWSET.

Index

A

Absolute time • 3–23
Access control list
  See ACL
ACL (access control list) • 6–1
Ada
  See VAX Ada
Address
  virtual memory • 5–10
Address space • 10–1
  allocating by page • 10–1, 10–3
  allocating in zones • 10–1
  deallocating by page • 10–1, 10–3
  zones • 10–1
Aligning data • 8–4
ALWAYS keyword
  GSMATCH option • 5–5
ANALYZE/RMS_FILE
  See Analyze/RMS_File Utility
ANALYZE/RMS_FILE command • 8–55
ANALYZE/RMS_File Utility (ANALYZE/RMS_FILE) • 1–38
APL
  See VAX APL
Arithmetic
  See also Condition handler
  using system routines • 1–24
Assembler • 1–9
AST (asynchronous system trap) • 4–7
  See also Synchronization
  delivery • 4–8
  execution • 4–7
  writing • 4–7
Asynchronous input/output • 7–47
Asynchronous system trap
  See AST

B

Barrier synchronization
  See Parallel processing
BASIC
  See VAX BASIC
Binary semaphore • 4–17
BLISS-32
  See VAX BLISS-32
Border
  virtual display • 7–10
Broadcast message • 7–43
  alternate handler • 7–44
  default handler • 7–43
Buffered input/output operation • 3–20

C

C
  See VAX C
Call frame condition handler • 9–13
Catchall handler • 9–5, 9–13
CDU (Command Definition Utility)
  command • 1–16
  creating command table • 1–17
  defining commands • 1–16
  modifying command table • 1–16
  parsing command • 1–17
Channel
  input/output • 7–45
CLUSTER option • 5–6
  See also linker
COBOL
  See VAX COBOL
Command Definition Utility
  See CDU
Common block • 3–6
  aligning • 8–4
  installing as a shared image • 5–13
  interprocess • 5–13
  modifying • 3–6
  per-process • 3–6
Common Data Dictionary • 1–8, 1–9, 1–10
Common event flag cluster
  permanent • 4–5
  temporary • 4–4
Communication
  intersystem • 3–26
Compilers • 1–5 to 1–11
  VAX Ada • 1–5

Compilers (cont'd.)
VAX BASIC • 1–6
VAX BLISS-32 • 1–6
VAX C • 1–7
VAX COBOL • 1–7
VAX DIBOL • 1–8
VAX FORTRAN • 1–8
VAX LISP • 1–8
VAX PASCAL • 1–9
VAX PL/I • 1–10
VAX RPG II • 1–10
VAX SCAN • 1–11
Composed input
See also Key table
terminating • 7–28
Condition code • 9–1
chaining • 9–23
defining • 9–7
modifying • 9–20
signaling • 9–5
SS$_EXQUOTA • 9–3
SS$_NOPRIV • 9–3
Condition code and message • 9–1
Condition handler
arithmetic • 9–26
call frame • 9–13
catchall • 9–13
condition code • 9–16
debugging • 9–20
establishing • 9–14
exiting • 9–17
last-chance exception vector • 9–13
mechanism array • 9–15
primary exception vector • 9–13
searching for • 9–12
secondary exception vector • 9–13
signal array • 9–14
traceback • 9–13
use of • 9–13, 9–20
writing • 9–14
Condition handling
default • 9–5
resignaling • 9–18
return status • 9–3
signal • 9–5
unwinding • 9–18
Control action
inhibit • 7–42
Control block
See also VMS RMS
See data structure
CONVERT
See Convert Utility
CONVERT/FDL command • 8–58
CONVERT/RECLAIM
See Convert/Reclaim Utility
Convert/Reclaim Utility (CONVERT/RECLAIM) • 1–39
Convert Utility (CONVERT) • 1–39
Counting semaphore • 4–17
CREATE/FDL
See Create/FDL Utility • 1–39
CREATE/FDL command • 8–57
Create/FDL Utility (CREATE/FDL) • 1–39
creating a data file • 8–57
CTRL/C • 7–33
CTRL/Y • 7–33
CTRL/Z • 7–5, 7–54
Current time • 3–23
Cursor movement • 7–20

D

Data
aligning • 8–4
interprocess • 5–13
sharing • 5–13
Database
compressing • 8–26
expanding • 8–32
record • 8–10
Data compression facility • 8–25
Data structure
FAB (file access block) • 1–36
NAM (name block) • 1–36
RAB (record access block) • 1–36
XAB (extended attribute block) • 1–36
DCL commands
ANALYZE/RMS_FILE command • 8–55
CONVERT/FDL • 8–58
CREATE/FDL command • 8–57
EDIT/FDL command • 8–55
DCX (Data/Expansion) routine • 8–25
Debuggers • 1–14 to 1–16
See Delta/Xdelta utility
See Symbolic Debugger
Debugging
condition handler • 9–20
exit handler • 9–30

/DELETE qualifier
LIBRARY command • 5–2
Delta time • 3–23
DELTA/XDELTA
See Delta/Xdelta Utility
Delta/Xdelta Utility (DELTA/XDELTA) • 1–15
Detached process
creating • 2–7
Device type • 7–50
DIBOL
See VAX DIBOL
Direct input/output operation • 3–20
Directive
See also Message Utility
.END • 9–8
.FACILITY • 9–7
.SEVERITY • 9–8
.TITLE • 9–9
Double-width characters
See also Screen management
See also Virtual display
specifying • 7–20
Dump file
See also SDA
analyzing • 1–21

E

Echo
terminal • 7–40
terminator • 7–24
EDIT/FDL
See Edit/FDL Utility • 1–39
EDIT/FDL command • 8–55
Edit/FDL Utility (EDIT/FDL) • 1–39
editor • 8–55
modifying a data file • 8–58
Editor
See Text processing
EDT editor • 1–3
EVE • 1–5
VAX Text Processing Utility • 1–4
EDT editor
mode
keypad • 1–3
line • 1–3
nokeypad • 1–4
EDT text editor
See EDT editor
.END directive • 9–8
End of file • 7–5
EQUAL keyword
GSMATCH option • 5–5
Error handling • 9–1
See Condition Handling
Escape sequence
read • 7–53
EVE editor
keypad emulation
EDT • 1–5
Numeric • 1–5
VT100 • 1–5
WPS • 1–5
Event flag • 4–1
See also synchronization
cluster • 4–1
common • 4–1
local • 3–2, 4–1
Event synchronization
see Synchronization
Exit
See also exit handler
image • 9–26
Exit handler • 7–53, 9–26
debugging • 9–30
establishing • 9–27
writing • 9–29
Extensible Vax Editor
See EVE editor
/EXTRACT qualifier
LIBRARY command • 5–2

F

FAB (file access block) • 1–36, 8–58
.FACILITY directive • 9–7
FAO argument
signaling • 9–12
FAO parameter
specifying • 9–12
/FAO_COUNT qualifier
Message Utility • 9–9
FDL$CREATE • 8–57
FDL$GENERATE • 8–55
FDL (File Definition Language) • 1–39, 8–54
applying source • 8–57
editor • 8–55
generating source • 8–55

FDL editor
  See Edit/FDL Utility (EDIT/FDL) • 8–55
FDL file • 1–39, 8–55
  creating • 8–55
  using existing • 8–55
File
  access strategies • 8–1
  attributes • 8–1, 8–3
  compressing • 8–26
  expanding • 8–32
  mapping • 8–4
  merging • 8–19
  modifying • 8–58
  sequential • 8–10
  sorting • 8–15
File management • 1–23
File terminator • 7–54
Flag
  See event flag
FORTRAN
  See VAX FORTRAN

G

Global section • 5–15
  multiprocessing • 4–18
  permanent • 5–19
  temporary • 5–19
  writable • 4–18
Global symbol • 5–11
  resolving • 5–11
  signaling with • 9–11
GSMATCH option • 5–6
  See also linker

H

Header
  library • 8–50
  library module • 8–48
Help library • 1–18
  displaying text • 8–52

I

I/O
  see Input/output
Identifier
  description • 6–1
If state
  composed input • 7–28
Image
  exiting • 9–26
  privileged • 6–2
  shareable • 5–3
IMAGELIB.OLB • 5–12
Image map
  See linker
Image run-down • 9–26
Input/output
  asynchronous • 7–47
  channel • 7–45
  checking device type • 7–50
  complex • 7–2
  device • 1–23
  echo • 7–40
  exit handler • 7–53
  file • 1–23
  lowercase • 7–42
  reading a single line • 7–4
  reading several lines • 7–5
  screen updates • 7–31
  simple • 7–1
  status of • 7–49
  synchronous • 7–46
  terminator • 7–4
    end of file • 7–54
    record • 7–53
  timeout • 7–41
  unsolicited input • 7–36
  uppercase • 7–42
  using SYS$QIO • 7–45, 7–49
  using SYS$QIOW • 7–45, 7–49
  writing simple character data • 7–6
Install
  privileged image • 6–2
Instruction
  interlocked • 4–18
  queue • 4–19
Interlocked instruction • 4–18
Interpreters

Interpreters (cont'd.)
  VAX APL • 1–6
  VAX BASIC • 1–6
  VAX LISP • 1–8
Interprocess communication • 3–7
  using mailboxes • 3–7
Intersystem communication • 3–26
Intraprocess communication • 3–1
  common blocks • 3–6
  global symbols • 3–6

K

Key
  See Sort/Merge Utility
Keypad
  reading from • 7–25
Key table
  reading from • 7–28

L

Last-chance exception vector • 9–13
LBR$CLOSE • 8–36
LBR$DELETE_DATA • 8–42
LBR$DELETE_KEY • 8–42
LBR$GET_HEADER • 8–50
LBR$GET_INDEX • 8–53
LBR$GET_RECORD • 8–43
LBR$INI_CONTROL • 8–36
LBR$INSERT_KEY • 8–40
LBR$LOOKUP_KEY • 8–40, 8–42, 8–43, 8–48
LBR$OPEN • 8–36
LBR$OUTPUT_HELP • 8–52
LBR$PUT_END • 8–40
LBR$PUT_RECORD • 8–40
LBR$REPLACE_KEY • 8–40
LBR$SET_MODULE • 8–48
LBR$_KEYNOTFND • 8–40
LEQUAL keyword
  GSMATCH option • 5–5
LIB$ADDX • 3–24
LIB$ADD_TIME • 3–24
LIB$CREATE_VM_ZONE • 10–1
LIB$DATE_TIME • 3–23
LIB$DAY • 3–25
LIB$DEC_OVER • 9–26
LIB$FLT_UNDER • 9–26
LIB$FREE_TIMER • 3–21
LIB$GETQUI • 3–22
LIB$GET_INPUT • 7–3
  example • 7–4
  obtaining several lines of input with • 7–5
  obtaining single line of input with • 7–4
  prompt • 7–4
LIB$GET_LUN • 7–3
LIB$GET_VM_PAGE • 10–1
LIB$INIT_TIMER • 3–20
LIB$INSERT_KEY • 8–45
LIB$INT_OVER • 9–26
LIB$MATCH_COND • 9–16
LIB$MULT_DELTA_TIME • 3–24
LIB$PUT_OUTPUT • 7–3
  example • 7–7
  writing simple output with • 7–6
LIB$SET_INDEX • 8–45
LIB$SHOW_TIMER • 3–20
LIB$SIGNAL
  invoking • 9–5
LIB$SIG_TO_RET
  establishing • 9–6
LIB$STAT_TIMER • 3–21
LIB$SUBX • 3–24
LIB$SUB_TIME • 3–24
LIBRARIAN
  See Librarian Utility
Library
  adding module with LBR routine • 8–40
  closing
    LBR$ routine • 8–36
  closing with LBR$ routine • 8–36
  compressing • 8–25
  creating with LBR routine • 8–36
  default object • 5–1
  deleting module with LBR routine • 8–42
  expanding • 8–25
  initializing with LBR routine • 8–36
  inserting module with LBR routine • 8–40
  listing index entries • 8–53
  macro • 5–3, 5–13
  message object module • 9–9
  module header • 8–48
  multiple indexes • 8–45
  multiple keys • 8–45
  object • 5–1, 5–12
    adding modules • 5–2
    creating • 5–2
    deleting a module • 5–2
    extracting a module • 5–2

Library
  object (cont'd.)
    listing modules • 5–2
    replacing modules • 5–2
    system default • 5–2
    user default • 5–2
  opening with LBR routine • 8–36
  processing index entries • 8–53
  processing index entry with LBR routine • 8–53
  replacing module • 8–40
  shareable image • 5–8
    adding • 5–8
    deleting • 5–8
    listing • 5–8
    replacing • 5–8
  system default • 5–12
  text • 5–3
  user default • 5–12
LIBRARY command • 1–19
  /CREATE qualifier • 5–2
  /DELETE qualifier • 5–2
  /EXTRACT qualifier • 5–2
  /LIST qualifier • 5–2
  /REPLACE qualifier • 5–2
Library module
  extracting with LBR routine • 8–43
Librarian Utility (LIBRARIAN)
  creating libraries • 1–17
  default logical names • 1–18
  library
    types of • 1–18
  LIBRARY command • 1–19
Line editing
  inhibit • 7–42
Linker • 1–11 to 1–13
  CLUSTER option • 5–6
  command qualifier summary • 1–13
  GSMATCH option • 5–5, 5–6
  image map • 1–13
  input • 1–12
  object language • 1–13
  options file • 1–13
  output • 1–12
  searching object libraries • 5–2
  UNIVERSAL option • 5–5
LINK/SHAREABLE command • 5–14
LISP
  See VAX LISP
/LIST qualifier
  LIBRARY command • 5–2
LNK$LIBRARY • 5–1
LNK$LIBRARY (cont'd.)
  See also Library
  See also Linker
Local symbol • 5–11
  signaling with • 9–11
Lock manager • 4–13
  See also synchronization
  queueing a lock request • 4–14

M

MACRO
  See VAX MACRO
Macro library • 1–18, 5–13
Mailbox • 3–7
  creating • 3–8
  input/output
    asynchronous • 3–9
    immediate • 3–9
    synchronous • 3–9
    using SYS$QIO • 3–9
    using SYS$QIOW • 3–9
  permanent • 3–8
  reading data from • 3–9
  temporary • 3–8
  writing data to • 3–9
mapped file • 8–4
Mapped file
  closing • 8–9
  saving • 8–9
Mathematical functions
  using system routines • 1–24
Mechanism array • 9–15
Memory management • 10–1
  using system routines • 1–23
  virtual memory • 1–23
Menu
  reading • 7–23
Menus
  creating with SMG$ routines • 7–22
MERGE command • 8–13
  file interface • 8–19
  record interface • 8–21
Message
  chaining • 9–23
  displaying • 9–22
  logging • 9–24
MESSAGE
  See Message Utility

Message text
  specifyling variables in • 9–9
Message Utility
  compiling message file • 9–9
Message Utility (MESSAGE) • 1–19, 9–7
  accessing message object module • 9–10
  creating a message object library • 9–10
  definition statements • 1–19
  directives • 1–19
  .END • 9–8
  .FACILITY • 9–8
  facility name • 9–8
  facility number • 9–8
  FAO parameters • 9–12
  /FAO_COUNT • 9–9
  logging messages • 9–24
  message object module • 9–9
  messages
    creating • 1–19
  message text • 9–9
  message text variables • 9–9
  modifying a message source file • 9–10
  .SEVERITY • 9–8
  source file • 1–19
  source module • 9–7
  .TITLE • 9–9
Modularity
  virtual displays • 7–31
Multiprocessing environment • 4–18
  scheduling • 4–19
  See also Synchronization • 4–18
Multistreamed workload • 4–18

N

NAM (name block) • 1–36
National Character Set Utility (NCS) • 1–22
NCS
  See National Character Set Utility
Network
  completing connection • 3–27
  connection request • 3–26
  exchanging messages • 3–28
  terminating connection • 3–30

O

Object language
  See linker
Object library • 1–18, 5–1, 5–12
  adding a module • 5–2
  creating • 5–2
  deleting a module • 5–2
  extracting a module • 5–2
  including message object module • 9–9
  listing modules • 5–2
  replacing a module • 5–2
Options
  creating with LBR$OPEN • 8–36
Options file • 5–8
  See also linker
  creating • 5–6

P

Page fault • 3–20
Parallel processing • 4–15
  initializing • 4–16
  subprocess
    creating • 4–16
    deleting • 4–16
  terminating • 4–16
  using semaphores • 4–17
  using spin locks • 4–16
Parallel programming • 4–18 to 4–19
PASCAL
  See VAX PASCAL
Pasteboard • 7–8
  creating • 7–9
  deleting • 7–9
  ID • 7–31
  sharing • 7–31
PATCH
  See Patch Utility
Patch Utility (PATCH) • 1–20
  input • 1–20
Per-process common blocks • 3–6
PL/I
  See VAX PL/I
PPL$CREATE_PROCESS • 4–16
PPL$ routines • 4–15

Primary exception vector • 9–13
Printer device width • 7–6
Privilege
  SS$_NOPRIV • 9–3
Privileged image
  installing • 6–2
Process
  communicating between • 3–7
  communicating within • 3–1
    using logical names • 3–2
    using symbols • 3–5
  creating • 2–1
  deleting • 2–15
  detached • 2–7
  execution • 2–14
  modes of execution • 2–1
  modifying name • 2–13
  obtaining information • 2–9
    using LIB$GETJPI • 2–9
    using SYS$GETJPI • 2–9
    using SYS$GETJPIW • 2–9
  priority
    modifying • 2–12
  privileges
    setting • 2–12
  scheduling • 2–12
Process management • 2–8
Processor
  synchronization • 4–18
Process rights list • 6–1
Program decomposition • 4–18
Program execution
  See also Synchronization
  specifying a time • 4–8, 4–9
  timed intervals • 4–10
Prompt for input
  with LIB$GET_INPUT • 7–4

Q

Queue information, obtaining • 3–22
Quotas
  SS$_EXQUOTA • 9–3

R

RAB (record access block) • 1–36, 8–58
Record
  compressing • 8–26
  expanding • 8–32
  I/O • 8–10
  merging • 8–21
  sorting • 8–16
Record management • 1–23
/REPLACE qualifier
  LIBRARY command • 5–2
Return status • 9–3
  from signal • 9–6
Rights database • 6–1
RMS
  See VMS RMS
$RMSDEF macro
  See also VMS RMS
RMS structures • 8–58
RMS utilities
  See VMS RMS
RPG II
  See VAX RPG II
Run-time library routine
  return status • 9–3
Run-Time Library routines • 1–24 to 1–29

S

SCAN
  See VAX SCAN
Screen management • 7–7
  See also Key table
  See also Pasteboard
  See also Video attribute
  See also Viewport
  See also Virtual keyboard
  deleting text • 7–21
  double-width characters • 7–19, 7–20
  drawing lines • 7–20
  inserting characters • 7–18
  menus
    creating • 7–22
    reading • 7–23
    types of • 7–22
  reading data • 7–23
  scrolling • 7–20
  See also Virtual display • 7–10
  setting background color • 7–9
  setting screen dimensions • 7–9
  using system routines • 1–23

Screen management (cont'd.)
video attributes • 7–20
viewport • 7–17
Scroll
backward • 7–19
down • 7–19
forward • 7–19
output • 7–19
up • 7–19
SDA (System Dump Analyzer) • 1–21 to 1–22
analyzing dump file • 1–21
Secondary exception vector • 9–13
Section
deleting • 8–9
global • 5–15
mapping • 8–4
private • 8–4
updating • 8–9
Security • 1–23
Semaphore • 4–17
See also Synchronization
binary • 4–17
counting • 4–17
Sequential file
creating • 8–10
merging • 8–13, 8–14
sorting • 8–13, 8–14
updating • 8–11
SETSWM • 10–4
.SEVERITY directive • 9–8
Shareable image • 5–3
adding • 5–8
contents of • 5–3
creating • 5–6
default file type • 5–9
default location • 5–9
deleting • 5–8
ID
major • 5–5
minor • 5–5
specifying major • 5–7
specifying minor • 5–7
library • 5–8
linking • 5–7, 5–8
listing • 5–8
replacing • 5–8
shared image • 5–10
specifying alternate locations • 5–9
transfer vector • 5–3, 5–6
universal symbol • 5–5
Shareable image library • 1–18
Shareable image library (cont'd.)
See also Shareable image
/SHAREABLE qualifier
LIBRARY command • 5–8
Shared files • 5–19
Shared image
creating • 5–10
Sharing Data
VMS RMS shared files • 5–19
Signal array • 9–14
Signaling • 9–5
changing to return status • 9–6
SMG$ADD_KEY_DEF • 7–28
SMG$CHANGE_VIRTUAL_DISPLAY • 7–15
SMG$CHECK_FOR_OCCLUSION • 7–12
SMG$CREATE_KEY_TABLE • 7–28
SMG$CREATE_PASTEBOARD • 7–8
SMG$CREATE_SUBPROCESS • 7–16
SMG$CREATE_VIRTUAL_DISPLAY • 7–8
SMG$CREATE_VIRTUAL_KEYBOARD • 7–24
SMG$DELETE_CHARS • 7–22
SMG$DELETE_LINE • 7–22
SMG$DELETE_PASTEBOARD • 7–9
SMG$DELETE_SUBPROCESS • 7–16
SMG$DELETE_VIRTUAL_DISPLAY • 7–14
SMG$DRAW_LINE • 7–20
SMG$DRAW_RECTANGLE • 7–20
SMG$ERASE_CHARS • 7–21
SMG$ERASE_COLUMN • 7–22
SMG$ERASE_DISPLAY • 7–21
SMG$ERASE_LINE • 7–21
SMG$ERASE_PASTEBOARD • 7–9
SMG$EXECUTE_COMMAND • 7–16
SMG$HOME_CURSOR • 7–17
SMG$INSERT_CHARS • 7–18
SMG$INSERT_LINE • 7–20
SMG$LABEL_BORDER • 7–10
SMG$LIST_PASTING_ORDER • 7–14
SMG$PASTE_VIRTUAL_DISPLAY • 7–8
SMG$POP_VIRTUAL_DISPLAY • 7–32
SMG$PUT_CHARS_HIGHWIDE • 7–19
SMG$PUT_LINE • 7–19
SMG$PUT_LINE_WIDE • 7–20
SMG$PUT_WITH_SCROLL • 7–19
SMG$READ_COMPOSED_LINE • 7–28
SMG$READ_FROM_DISPLAY • 7–23
SMG$READ_STRING • 7–24
SMG$RESTORE_PHYSICAL_SCREEN • 7–31
SMG$RETURN_CURSOR_POS • 7–18
SMG$SAVE_PHYSICAL_SCREEN • 7–31
SMG$SCROLL_DISPLAY_AREA • 7–20

SMG$SET_CURSOR_ABS • 7–17
SMG$SET_CURSOR_REL • 7–17
SMG$SET_DISPLAY_SCROLL_REGION • 7–20
SMG$SET_PHYSICAL_CURSOR • 7–18
SMG$UNPASTE_VIRTUAL_DISPLAY • 7–14
SOR$BEGIN_MERGE • 8–19
SOR$BEGIN_SORT • 8–15
SOR$END_SORT • 8–15
SOR$PASS_FILES • 8–15, 8–19
SOR$RELEASE_REC • 8–16
SOR$RETURN_REC • 8–16
SOR$SORT_MERGE • 8–15
SORT
  See Sort/Merge Utility
SORT command • 8–13
  file interface • 8–15
  record interface • 8–16
Sort/Merge routine
  See SOR routine
Sort/Merge Utility (SORT) • 8–13
  file interface • 8–14, 8–15, 8–19
  keys • 8–14
  multiple sort operations • 8–14
  record interface • 8–14, 8–16, 8–21
Spawned subprocess
  See Subprocess
Spin locks • 4–16
  See also Synchronization
STARLET.OLB • 5–1, 5–12
Subprocess
  creating
    with LIB$SPAWN • 2–2
    with PPL$ routines • 4–16
    with SMG$ routines • 7–16
    with SYS$CREPRC • 2–3
  creatingz
    with PPL$CREATE_PROCESS • 2–4
  deleting with PPL$ routines • 4–16
  priority
    setting • 2–12
  program debugging • 2–5
SUMSLP
  See SUMSLP Utility • 1–20
SUMSLP Utility (SUMSLP) • 1–20 to 1–21
Swap mode
  changing • 10–4
Symbol
  defining • 5–11
  global • 5–11
  local • 5–11
  referring to • 5–10
Symbol (cont'd.)
  storage • 5–10
  universal • 5–5
  unresolved • 5–12
Symbolic Debugger • 1–14 to 1–15
Synchronization • 1–24
  barrier • 4–17
  passing control to another image • 4–19
  using asynchronous system traps • 4–7
  using detached processes • 4–8
  using events flags • 4–1
  using process priority • 4–19
  using semaphores with PPL$ routines • 4–17
  using spin locks with PPL$ routines • 4–16
  using subprocesses • 4–8
Synchronization with parallel processing routines
  See Parallel processing
Synchronous input/output • 7–46
SYS$ASCTIM • 3–24
SYS$ASSIGN • 7–45
SYS$BINTIM • 3–24
SYS$CREATE • 8–8
SYS$CREMBX • 3–8
SYS$CRETVA • 10–3
SYS$CRMPSC • 8–4, 8–5
SYS$DASSGN • 8–9
SYS$DCLEXH • 9–27
SYS$DELTVA • 8–9
SYS$ERROR • 9–24
SYS$EXPREG • 10–3
SYS$FAO • 3–24
SYS$GETDVI • 7–50
SYS$GETQUI • 3–22
SYS$GETSYI • 3–22
SYS$GETTIM • 3–24
SYS$INPUT • 9–24
  default value of • 7–2
  redefining value of • 7–3
  using with LIB$GET_INPUT • 7–3
  using with LIB$PUT_OUTPUT • 7–3
SYS$LCKPAG • 10–4
SYS$LKWSET • 10–3
SYS$MGBLSC • 5–15
SYS$OPEN • 8–8
SYS$OUTPUT
  default value of • 7–2
  redefining value of • 7–3
  using with LIB$GET_INPUT • 7–3
  using with LIB$PUT_OUTPUT • 7–3
SYS$OUTPUT_HELP • 8–36
SYS$PUTMSG • 9–15, 9–22

SYS$QIO • 7–45
SYS$QIOW • 7–45
SYS$SETEXV • 9–13
SYS$SHARE • 5–9
SYS$ULKPAG • 10–4
SYS$ULWSET • 10–4
SYS$UNWIND • 9–18
SYS$UPDSEC • 8–9
System Dump Analyzer
  See SDA
System information
  See also timer statistics
System routines • 1–22 to 1–24
  system services
    asynchronous • 4–12
    synchronous • 4–12
Systems
  communication between • 3–26
System service • 1–29
  return status • 9–3
System time • 3–23
System timer
  cancelling • 4–12
  setting • 4–11

T

Terminal characteristics • 7–51
Terminal device width • 7–6
Terminal echo • 7–40
  disabling • 7–41
Terminal timeout • 7–41
Terminator
  See Input/output
  echo • 7–24
  file • 7–54
  record • 7–53
Text library • 1–18
Text processing • 1–3
Text Processing
  EVE editor • 1–5
Time • 3–23
  See also absolute time
  See also current
  See also delta time
  internal format • 3–23
  obtaining
    using SYS$ASCTIM • 3–24
    using SYS$BINTIM • 3–24
Time
  obtaining (cont'd.)
    using SYS$FAO • 3–24
    using SYS$GETTIM • 3–24
Time manipulation • 3–24
  converting • 3–24
  formatting • 3–24
  using LIB$ADDX • 3–24
  using LIB$ADD_TIME • 3–24
  using LIB$DAY • 3–25
  using LIB$MULT_DELTA_TIME • 3–24
  using LIB$SUBX • 3–24
  using LIB$SUB_TIME • 3–24
Timer
  deallocating • 3–21
  initializing • 3–20
  obtaining statistics • 3–20, 3–21
  statistics
    buffer input/output • 3–20
    CPU time • 3–20
    direct input/output • 3–20
    elapsed time • 3–20
    page fault • 3–20
TITLE directive • 9–9
TPU
  See VAXTPU
Traceback handler • 9–5, 9–13
Transfer vector • 5–3
  See also Shareable image
  compiling • 5–6
  creating • 5–6
  deleting • 5–4
  placement of • 5–3
  reasons for using • 5–4
TRM$M_TM_ESCAPE • 7–25
TRM$M_TM_NOECHO • 7–25
TRM$M_TM_TRMNOECHO • 7–24
Type-ahead buffer • 7–39

U

UFO
  see User-file open
UNIVERSAL option
  See Linker
Universal symbol • 5–5
  resolving • 5–5
Unwind condition handler • 9–18

User-defined condition code
  signaling • 9–10
User-file open • 8–8
User-open routine • 8–58
Utilities
  see entries for each utility
  invoking from a program • 1–24
Utility Routines • 1–34

V

VAX Ada • 1–5
VAX APL • 1–6
VAX BASIC • 1–6
VAX BLISS-32 • 1–6
VAX C • 1–7
VAX COBOL • 1–7
VAX common language environment • 1–5
VAX compilers
  See compilers
VAX DIBOL • 1–8
VAX FORTRAN • 1–8
VAX LISP • 1–8
VAX MACRO • 1–9
VAX PASCAL • 1–9
VAX PL/I • 1–10
VAX RPG II • 1–10
VAX SCAN • 1–11
VAX Text Processing Utility
  See VAXTPU
VAXTPU (VAX Text Processing Utility) • 1–4
  EVE editor • 1–5
Video attribute • 7–10, 7–16
  current • 7–16
  default • 7–16
Video attributes • 7–20
Viewport • 7–17
Virtual display • 7–10
  See also Viewport
  checking occlusion of • 7–12
  creating • 7–10
  creating a subprocess from • 7–16
  cursor movement • 7–20
  deleting • 7–14
  deleting text • 7–21
  drawing lines • 7–20
  erasing • 7–14
  ID • 7–10, 7–32
  inserting text • 7–18, 7–20
Virtual display (cont'd.)
  list pasting order of • 7–14
  logical cursor position • 7–17
  modifying • 7–15
  obtaining the pasting order • 7–14
  overwriting text • 7–18, 7–20
  pasting • 7–11
  physical cursor position • 7–18
  popping • 7–15
  reading data from • 7–23
  rearranging • 7–13
  scrolling • 7–20
  sharing • 7–32
  specifying double-width characters • 7–20
  specifying video attributes • 7–10
  viewport • 7–17
  writing double-width characters • 7–19
  writing text to • 7–17
Virtual keyboard
  reading data from • 7–23, 7–24
VMS RMS (Record Management Services) • 1–35 to 1–38
  Analyze/RMS_File Utility • 1–38
  control block
    FAB • 1–36
    NAM • 1–36
    XAB • 1–36
  Convert/Reclaim Utility • 1–39
  Convert Utility • 1–39
  Create/FDL Utility • 1–39
  device support • 1–36
  Edit/FDL Utility • 1–39
  macro • 1–37
  macros • 1–37

W

Working set
  adjusting size • 10–3
  locking pages • 10–3

X

XAB (extended attribute block) • 1–36

Reader's Comments

Guide to VMS Programming Resources
AA–LA57A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page Description

____ ______________________________

____ ______________________________

____ ______________________________

____ ______________________________

____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ____________

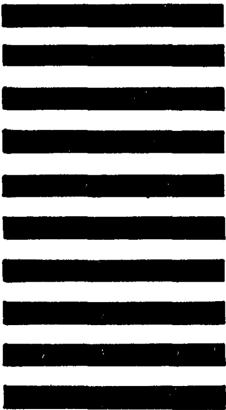
Company ______________________________ Date ____________

Mailing Address ______________________________

______________________________ Phone ____________

-- Do Not Tear - Fold Here and Tape

digital™



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here