Guide to Programming on VAX/VMS

Order Number: AI–Y503B–TE

April 1986

This guide contains practical guidelines for using VAX/VMS program development tools. Although this manual has a FORTRAN orientation and assumes a reading knowledge of the language, the guidelines and concepts described can be implemented in other programming languages. To help illustrate key concepts, many examples have been included.

Revision/Update Information:	This document supersedes the *Guide to Programming on VAX/VMS (FORTRAN Edition)* Version 4.0.
Software Version:	VAX/VMS Version 4.4

digital equipment corporation
maynard, massachusetts

April 1986

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.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

PREFACE xvii

NEW AND CHANGED FEATURES xxi

CHAPTER 1 INTERUNIT LOGIC 1-1

1.1 INVOKING PROGRAM UNITS 1-2
1.1.1 Subroutine 1-3
1.1.2 Function 1-4
1.1.3 Intrinsic Subprograms 1-5
1.1.4 System-Defined Procedures 1-6

1.2 TRANSFER OF DATA 1-7
1.2.1 Argument Lists 1-8
1.2.2 Common Blocks 1-10
1.2.2.1 Primary Uses • 1-10
1.2.2.2 Initialization • 1-12

1.3 PASSING CONTROL INFORMATION AMONG USER SUBPROGRAMS 1-13
1.3.1 Flags 1-13
1.3.2 Masks 1-14
1.3.3 Indicative Values 1-15

1.4 PASSING VARIABLE-LENGTH DATA 1-16
1.4.1 Character Strings 1-16
1.4.2 Arrays 1-17

1.5 PASSING ARGUMENTS TO SYSTEM-DEFINED PROCEDURES 1-20
1.5.1 Mechanics of Passing Arguments 1-22
1.5.2 Aligning Data 1-23
1.5.3 Passing Bytes 1-24
1.5.4 Passing Words 1-24

1.5.5 **Passing Longwords** 1-24
- 1.5.5.1 Specifying Byte and Word Values • 1-25
- 1.5.5.2 Setting Bits • 1-26

1.5.6 **Passing Quadwords** 1-28
- 1.5.6.1 Specifying Byte, Word, and Longword Values • 1-28
- 1.5.6.2 Setting Bits • 1-30

1.5.7 **Passing Variable-Length Data Structures** 1-31

1.6 READING INFORMATION RETURNED FROM SYSTEM-DEFINED PROCEDURES 1-34

1.6.1 **Masks** 1-35

1.6.2 **Addresses** 1-36
- 1.6.2.1 Numeric Data • 1-36
- 1.6.2.2 Character Data • 1-36

1.6.3 **Buffers** 1-38

CHAPTER 2 INTRAUNIT LOGIC AND LOCAL STORAGE 2-1

2.1 LOCAL STORAGE 2-3

2.1.1 **Data Type Definition** 2-3

2.1.2 **Variable Initialization** 2-5
- 2.1.2.1 One Time Initialization • 2-5
- 2.1.2.2 Reinitialization • 2-6

2.1.3 **Named Constants** 2-6

2.1.4 **Equivalent Variables** 2-7

2.1.5 **Contiguous Storage** 2-8

2.1.6 **Large Data Structures and Dynamic Storage** 2-9
- 2.1.6.1 Dynamic Storage of Numeric Data • 2-10
- 2.1.6.2 Dynamic Storage of Character Data • 2-11
- 2.1.6.3 Dynamic Storage Statistics • 2-13

2.2 SERIAL EXECUTION 2-13

2.3 CONDITIONAL EXECUTION 2-14

2.3.1 **Specifying the Condition** 2-14

2.3.2 **Single Conditional Block** 2-15

2.3.3 **Multiple Conditional Blocks** 2-16
- 2.3.3.1 Exclusive Conditional • 2-16
- 2.3.3.2 Inclusive Conditional • 2-18

2.4 ITERATIVE LOGIC 2-19
2.4.1 DO WHILE Statement 2-21
2.4.2 Indexed DO Statement 2-21

2.5 NESTING 2-22

CHAPTER 3 PROGRAM SYNCHRONIZATION AND COMMUNICATION 3-1

3.1 CREATING PROCESSES 3-1
3.1.1 Creating Subprocesses 3-2
3.1.1.1 Invoking LIB$SPAWN • 3-2
3.1.1.2 Subprocess Context • 3-3
3.1.1.3 Subprocess Execution • 3-4
3.1.1.4 Debugging a Program in a Subprocess • 3-5
3.1.2 Synchronizing Terminal I/O with the Lock Manager 3-5
3.1.3 Creating Detached Processes 3-7
3.1.4 Specifying a Time for Program Execution 3-9
3.1.4.1 Specified Time • 3-9
3.1.4.2 Timed Intervals • 3-10

3.2 EXAMINING AND MODIFYING PROCESSES 3-12
3.2.1 Examining Processes 3-12
3.2.2 Monitoring Program Execution 3-16
3.2.3 Controlling Process Scheduling 3-17

3.3 CONTROLLING PROGRAM EXECUTION 3-18
3.3.1 Synchronizing Operations with Event Flags 3-18
3.3.1.1 Manipulating Event Flags • 3-18
3.3.1.2 Common Event Flags • 3-20
3.3.1.3 Synchronous and Asynchronous System Services • 3-22
3.3.2 Interrupting Execution with an AST 3-24

3.4 INTERPROCESS COMMUNICATION 3-25
3.4.1 Symbols and Logical Names 3-25
3.4.2 Mailboxes 3-29
3.4.2.1 Synchronous Mailbox I/O • 3-31
3.4.2.2 Immediate Mailbox I/O • 3-34
3.4.2.3 Asynchronous Mailbox I/O • 3-39

3.4.3 **Sharing Data** 3-42
3.4.3.1 Installed Common Blocks • 3-43
3.4.3.2 Global Sections • 3-45

3.5 **INTRAPROCESS COMMUNICATION** 3-50
3.5.1 **Per-Process Common Blocks** 3-50
3.5.2 **Passing Control** 3-52

3.6 **INTERSYSTEM COMMUNICATION** 3-53
3.6.1 **Requesting a Network Connection** 3-53
3.6.2 **Completing a Network Connection** 3-55
3.6.3 **Exchanging Messages** 3-55
3.6.3.1 Terminating a Network Connection • 3-58

CHAPTER 4 IMPLEMENTATION CYCLE 4-1

4.1 **COMPILING AND LINKING PROGRAMS** 4-1
4.1.1 **Creating Source Programs** 4-2
4.1.2 **Compiling Programs** 4-4
4.1.2.1 Development Systems • 4-5
4.1.2.2 Production Systems • 4-6
4.1.2.3 Errors • 4-6
4.1.3 **Object Libraries** 4-8
4.1.4 **Linking and Executing Programs** 4-8
4.1.5 **Privileged Programs** 4-10
4.1.6 **Building Programs** 4-11
4.1.6.1 Build Process • 4-11
4.1.6.2 System Integrity • 4-12
4.1.6.3 Checkin and Checkout • 4-15

4.2 **SYMBOLS** 4-18
4.2.1 **Defining Symbols** 4-19
4.2.2 **Local and Global Symbols** 4-19
4.2.3 **Referencing Global Symbols** 4-20
4.2.4 **Resolving Global Symbols** 4-21
4.2.4.1 Explicitly Named Modules and Libraries • 4-21
4.2.4.2 System Default Libraries • 4-22
4.2.4.3 User Default Libraries • 4-22
4.2.4.4 Macro Libraries • 4-23
4.2.5 **FORTRAN Definition Libraries and Files** 4-24

4.3 SHAREABLE IMAGES 4-25
4.3.1 Creating Shareable Images ______ 4-25
4.3.1.1 Transfer Vectors • 4-28
4.3.1.2 GSMATCH Option • 4-30
4.3.1.3 UNIVERSAL option • 4-30
4.3.2 Shareable Image Libraries ______ 4-31
4.3.3 Linking Shareable Images ______ 4-31
4.3.4 Shared Shareable Images ______ 4-33

4.4 LISTINGS 4-33
4.4.1 Compiler Listings ______ 4-33
4.4.1.1 Source Program Listing • 4-34
4.4.1.2 Machine Code Listing • 4-35
4.4.1.3 Storage Map • 4-36
4.4.1.4 Compilation Summary • 4-39
4.4.2 Image Maps ______ 4-40

CHAPTER 5 USING THE DEBUGGER 5-1

5.1 INVOKING AND TERMINATING THE DEBUGGER 5-2
5.1.1 Invoking the Debugger ______ 5-2
5.1.2 Interrupting the Debugger ______ 5-3
5.1.3 Terminating the Debugger ______ 5-4

5.2 ENTERING DEBUGGER COMMANDS 5-4
5.2.1 Using Debugger HELP ______ 5-4
5.2.2 Abbreviating Debugger Commands ______ 5-4
5.2.3 Using the Keypad ______ 5-5
5.2.3.1 Default Key Definitions • 5-5
5.2.3.2 User Key Definitions • 5-7

5.3 USING SCREEN DISPLAYS 5-7
5.3.1 Invoking and Terminating Screen Mode ______ 5-8
5.3.2 Defining Windows ______ 5-10
5.3.3 Manipulating Displays ______ 5-10
5.3.3.1 Showing Displays • 5-10
5.3.3.2 Removing Displays • 5-11
5.3.3.3 Scrolling Displays • 5-12
5.3.3.4 Using Pseudodisplay Names • 5-13
5.3.4 Creating Displays ______ 5-13

5.4 CONTROLLING PROGRAM EXECUTION 5-14

5.4.1 Starting Program Execution 5-15

5.4.1.1 The GO Command • 5-15

5.4.1.2 STEP Command • 5-15

5.4.1.3 CALL Command • 5-17

5.4.1.4 SHOW CALLS Command • 5-17

5.4.2 Suspending (or Tracing) Program Execution 5-18

5.4.2.1 Breakpoints and Tracepoints • 5-18

5.4.2.2 Watchpoints • 5-20

5.4.3 Displaying Source Lines 5-21

5.4.4 Using Logical Control Structures 5-22

5.5 SYMBOLIC DEBUGGING 5-23

5.5.1 Maintaining Symbol Information 5-23

5.5.2 Referencing Symbols 5-25

5.5.3 Resolving Symbol References 5-26

5.5.3.1 Path-Name Prefix • 5-27

5.5.3.2 SET SCOPE Command • 5-28

5.5.4 Defining Symbols 5-28

5.5.5 Displaying Symbol Information 5-29

5.6 MANIPULATING DATA 5-30

5.6.1 Displaying Values 5-31

5.6.2 Calculating Values 5-32

5.6.3 Assigning Values 5-33

5.6.4 Specifying Data Type 5-33

5.6.5 Specifying Radix 5-34

5.7 USING COMMAND PROCEDURES 5-35

5.7.1 Displaying Commands 5-36

5.7.2 Passing Values 5-36

5.7.3 Initialization Files 5-37

5.8 USING LOG FILES 5-38

CHAPTER 6 DATA STRUCTURES 6-1

6.1 DEFINITION AND REFERENCE OF DATA ITEMS 6-1

6.2 ASSIGNMENT OF VALUES TO VARIABLES 6-3

6.3 NUMERIC DATA 6-4
6.3.1 Bytes 6-4
6.3.2 Integers 6-4
6.3.3 Real Numbers 6-5
6.3.4 Complex Numbers 6-6

6.4 NUMERIC OPERATIONS 6-7
6.4.1 Arithmetic Operations 6-7
6.4.2 Relational Operations 6-8
6.4.3 System Arithmetic Routines 6-9
6.4.4 Arithmetic Errors 6-10

6.5 LOGICAL DATA 6-10

6.6 CHARACTER DATA 6-12
6.6.1 Defining Character Strings 6-12
6.6.2 Character Constants 6-13
6.6.3 Referencing Character Strings 6-13
6.6.4 Character String Operations 6-13
6.6.4.1 Padding and Truncation • 6-14
6.6.4.2 Concatenation • 6-14
6.6.4.3 Intrinsic Character Functions • 6-15
6.6.5 Nonprintable Characters 6-15
6.6.6 Counted Strings 6-15

6.7 UNTYPED DATA 6-16
6.7.1 Untyped Constants 6-16
6.7.1.1 Hexadecimal Constants • 6-17
6.7.1.2 Octal Constants • 6-18
6.7.1.3 Hollerith Constants • 6-19
6.7.2 Bit Manipulation 6-19

6.8 CONVERSION BETWEEN DATA TYPES 6-20
6.8.1 Numeric Conversions 6-20
6.8.2 Formatted Conversions 6-21
6.8.3 Automatic Conversions 6-22

6.9 ARRAYS 6-23
6.9.1 Defining Arrays 6-23
6.9.2 Referencing Arrays 6-24
6.9.2.1 Per-Element Processing • 6-24
6.9.2.2 Multielement Processing • 6-25
6.9.2.3 Full Array Processing • 6-26
6.9.3 Storage and Bounds Considerations 6-27
6.9.4 Multidimensional Arrays 6-28
6.9.4.1 Storage of Multidimensional Arrays • 6-28
6.9.4.2 Processing Multidimensional Arrays • 6-29

6.10 RECORDS 6-31
6.10.1 Defining Record Formats 6-32
6.10.1.1 Typed Data • 6-33
6.10.1.2 Substructures • 6-34
6.10.1.3 Unions • 6-35
6.10.2 Declaring Record Variables 6-36
6.10.3 Referencing Records 6-37
6.10.4 Storing Record Structures 6-39

6.11 SYSTEM INFORMATION 6-40
6.11.1 Timer Statistics 6-40
6.11.2 System Time 6-43
6.11.2.1 Current Time • 6-44
6.11.2.2 Time Manipulation • 6-44
6.11.3 Emulated Instructions 6-46

CHAPTER 7 COMMAND INPUT AND SYNTAX ANALYSIS 7-1

7.1 COMMAND DESCRIPTION 7-1
7.1.1 Command Name and Image 7-1

7.1.2 **Parameters** — 7-2
7.1.2.1 Labels • 7-2
7.1.2.2 Required Parameters • 7-3
7.1.2.3 Values • 7-3
7.1.2.4 Prompting • 7-4
7.1.3 **Qualifiers** — 7-4
7.1.3.1 Negatable Qualifiers • 7-5
7.1.3.2 Default Presence • 7-5
7.1.3.3 Placement • 7-6
7.1.3.4 Values • 7-6
7.1.4 **Value Types** — 7-7
7.1.4.1 Built-in Value Types • 7-7
7.1.4.2 Keywords • 7-7
7.1.5 **Disallowing Entities and Combinations** — 7-9
7.1.6 **Syntax Changes** — 7-9
7.1.7 **Keyword and Definition Paths** — 7-10
7.1.7.1 Keyword Paths • 7-10
7.1.7.2 Definition Paths • 7-10

7.2 **COMMAND SETUP** 7-11
7.2.1 **Process Command Table** — 7-11
7.2.2 **DCL Command Table** — 7-12
7.2.3 **User Command Table** — 7-12
7.2.3.1 Copying the DCL Command Table • 7-12
7.2.3.2 Establishing the User Table • 7-13
7.2.3.3 Creating a Table from Scratch • 7-13
7.2.4 **Deleting Commands** — 7-14

7.3 **SYNTAX ANALYSIS** 7-14
7.3.1 **Checking for the Presence of Elements** — 7-15
7.3.2 **Getting Element Values** — 7-18
7.3.3 **Determining the Position of a Qualifier** — 7-20
7.3.4 **Examining the Command Line and Verb** — 7-22

7.4 **SUBCOMMANDS AND INTERNAL PARSING** 7-22
7.4.1 **Defining Subcommands** — 7-22
7.4.2 **Parsing Subcommands** — 7-23
7.4.3 **Dispatching to a Subprogram** — 7-26
7.4.4 **Returning to the Invoking Command** — 7-28

CHAPTER 8 USER INPUT/OUTPUT 8-1

8.1 CONVERSATIONAL I/O 8-2
- 8.1.1 Device Selection 8-2
  - 8.1.1.1 Implied Unit for FORTRAN I/O • 8-3
  - 8.1.1.2 Explicit Unit for FORTRAN I/O • 8-4
- 8.1.2 Getting a Line of Input 8-5
  - 8.1.2.1 LIB$GET_INPUT • 8-5
  - 8.1.2.2 ACCEPT Statement • 8-6
  - 8.1.2.3 READ Statement • 8-6
- 8.1.3 Getting Many Lines of Input 8-8
- 8.1.4 Writing Output 8-10
  - 8.1.4.1 LIB$PUT_OUTPUT • 8-10
  - 8.1.4.2 WRITE Statement • 8-11
  - 8.1.4.3 Formatting Numeric Output • 8-12
  - 8.1.4.4 Multiple Output Records • 8-13
  - 8.1.4.5 TYPE Statement • 8-13
  - 8.1.4.6 Processing Arrays and Records • 8-13
  - 8.1.4.7 Carriage Control • 8-14

8.2 SCREEN MANAGEMENT 8-15
- 8.2.1 Pasteboards 8-17
- 8.2.2 Virtual Displays 8-18
  - 8.2.2.1 Creating a Virtual Display • 8-18
  - 8.2.2.2 Pasting Virtual Displays • 8-19
  - 8.2.2.3 Rearranging Virtual Displays • 8-21
  - 8.2.2.4 Removing Virtual Displays • 8-23
  - 8.2.2.5 Modifying a Virtual Display • 8-24
- 8.2.3 Writing 8-25
  - 8.2.3.1 Positioning the Cursor • 8-26
  - 8.2.3.2 Writing Data Character by Character • 8-26
  - 8.2.3.3 Writing Data Line by Line • 8-28
  - 8.2.3.4 Drawing Lines • 8-29
  - 8.2.3.5 Deleting Text • 8-30
- 8.2.4 Reading 8-31
  - 8.2.4.1 Reading from a Display • 8-31
  - 8.2.4.2 Reading from a Virtual Keyboard • 8-32
  - 8.2.4.3 Reading from the Keypad • 8-34
  - 8.2.4.4 Reading Composed Input • 8-37
- 8.2.5 Controlling Screen Updates 8-39
- 8.2.6 Modularity 8-40

8.3 SPECIAL INPUT/OUPUT ACTIONS 8-42
8.3.1 CTRL/C and CTRL/Y Interrupts 8-42
8.3.2 Unsolicited Input 8-46
8.3.3 Type-Ahead Buffer 8-48
8.3.4 Echo 8-50
8.3.5 Timeout 8-51
8.3.6 Lowercase to Uppercase Conversion 8-52
8.3.7 Line Editing and Control Actions 8-52
8.3.8 Broadcasts 8-53

8.4 SYS$QIO AND SYS$QIOW SYSTEM SERVICES 8-55
8.4.1 Read Operations 8-55
8.4.2 Write Operations 8-59
8.4.3 Checking the Device Type 8-61
8.4.4 Terminal Characteristics 8-62
8.4.5 Record Terminators 8-64

CHAPTER 9 FILE INPUT/OUTPUT 9-1

9.1 FILE OPERATIONS 9-2
9.1.1 File Attributes 9-2
9.1.1.1 Logical Unit Number • 9-2
9.1.1.2 File Name • 9-3
9.1.1.3 File Organization and Access • 9-3
9.1.1.4 Record Structure • 9-4
9.1.1.5 File I/O • 9-5
9.1.1.6 File Status and Disposition • 9-6
9.1.1.7 Protection and Access • 9-7
9.1.1.8 Storage Allocation • 9-8
9.1.2 Opening Files 9-9
9.1.2.1 Opening a New or an Existing File • 9-9
9.1.2.2 Opening a File of Unknown Status • 9-10
9.1.2.3 Opening a File Across the Network • 9-11
9.1.3 Choosing Keywords for I/O Statements 9-11
9.1.4 Repositioning Within a File 9-14
9.1.5 Getting Information About a File 9-14
9.1.6 Closing a File 9-15

9.2 LOADING AND UNLOADING A DATABASE 9-16
9.2.1 FORTRAN I/O 9-16

9.2.2 SYS$CRMPSC — 9-19
9.2.2.1 Mapping a File • 9-20
9.2.2.2 User-Open Routine • 9-23
9.2.2.3 Initializing a Mapped Database • 9-25
9.2.2.4 Saving a Mapped File • 9-25

9.3 PER-RECORD PROCESSING OF ENTIRE DATABASE 9-26
9.3.1 Creating a Sequential File — 9-27
9.3.2 Updating a Sequential File — 9-28
9.3.3 Sorting and Merging Sequential Files — 9-30
9.3.3.1 Passing Key Information • 9-31
9.3.3.2 Sorting with the File Interface • 9-32
9.3.3.3 Sorting with the Record Interface • 9-33
9.3.3.4 Merging with the File Interface • 9-37
9.3.3.5 Merging with the Record Interface • 9-38

9.4 PROCESSING PARTS OF THE DATABASE 9-43
9.4.1 Creating an Indexed File — 9-43
9.4.2 Writing to an Indexed File — 9-45
9.4.3 Accessing a Record Directly — 9-47
9.4.4 Indexed Sequential Access Method (ISAM) — 9-49
9.4.5 Accessing a Record Using Multiple Keys — 9-52
9.4.6 Updating a Record in an Indexed File — 9-56
9.4.7 Deleting a Record from an Indexed File — 9-57
9.4.8 Handling Duplicate Keys — 9-58

9.5 DATA COMPRESSION AND EXPANSION 9-59
9.5.1 Compression Procedures — 9-60
9.5.2 Expansion Procedures — 9-67

9.6 LIBRARY PROCEDURES 9-71
9.6.1 Creating, Opening, and Closing Libraries — 9-72
9.6.2 Adding Modules — 9-76
9.6.3 Deleting Modules — 9-79
9.6.4 Extracting Modules — 9-80
9.6.5 Using Multiple Keys and Multiple Indexes — 9-83
9.6.6 Accessing Module Headers — 9-86
9.6.7 Reading Library Headers — 9-88
9.6.8 Displaying Help Text — 9-89
9.6.9 Listing and Processing Index Entries — 9-91

9.7 FILE DEFINITION LANGUAGE 9-92
- 9.7.1 Creating an FDL File 9-93
  - 9.7.1.1 Using the FDL Editor • 9-93
  - 9.7.1.2 Using the Characteristics of an Existing Data File • 9-94
- 9.7.2 Applying an FDL File to a Data File 9-95
  - 9.7.2.1 Creating a New Data File • 9-96
  - 9.7.2.2 Modifying an Existing Data File • 9-96

9.8 USER-OPEN ROUTINES 9-97
- 9.8.1 USEROPEN Specifier 9-98
- 9.8.2 Writing a User-Open Routine 9-98
- 9.8.3 Setting FAB and RAB Fields 9-100

CHAPTER 10 RUN-TIME ERRORS 10-1

10.1 GENERAL ERROR HANDLING 10-1
- 10.1.1 Condition Code and Message 10-2
- 10.1.2 Return Status Convention 10-3
  - 10.1.2.1 Testing Returned Condition Codes • 10-3
  - 10.1.2.2 Testing SS$_NOPRIV and SS$_EXQUOTA • 10-4
- 10.1.3 Signaling Mechanism 10-6
  - 10.1.3.1 Default Condition Handling • 10-6
  - 10.1.3.2 Changing a Signal to a Return Status • 10-7

10.2 DEFINING CONDITION CODES AND MESSAGES 10-9
- 10.2.1 Creating the Source File 10-9
  - 10.2.1.1 Specifying Facility • 10-10
  - 10.2.1.2 Specifying Severity • 10-11
  - 10.2.1.3 Specifying Condition Names and Messages • 10-11
  - 10.2.1.4 Specifying Variables in the Message Text • 10-12
- 10.2.2 Compiling and Linking the Messages 10-12
- 10.2.3 Signaling User-Defined Codes and Messages 10-13
  - 10.2.3.1 Signaling with Global Symbols • 10-14
  - 10.2.3.2 Signaling with Local Symbols • 10-14
  - 10.2.3.3 Specifying FAO Parameters • 10-15

10.3 FORTRAN I/O ERRORS 10-16
10.3.1 Unexpected FORTRAN I/O Errors 10-16
10.3.2 Expected FORTRAN I/O Errors 10-17

10.4 CONDITION HANDLERS 10-18
10.4.1 Establishing a Condition Handler 10-21
10.4.2 Writing a Condition Handler 10-21
10.4.2.1 Dummy Arguments • 10-21
10.4.2.2 Checking the Condition Code • 10-23
10.4.2.3 Exiting • 10-24
10.4.3 Debugging 10-27
10.4.4 Condition Handler Functions 10-28
10.4.4.1 Modifying Condition Codes • 10-28
10.4.4.2 Displaying Messages • 10-29
10.4.4.3 Chaining Messages • 10-30
10.4.4.4 Logging Messages • 10-32
10.4.5 System-Defined Arithmetic Condition Handlers 10-34

10.5 EXIT HANDLERS 10-35
10.5.1 Establishing an Exit Handler 10-36
10.5.2 Writing an Exit Handler 10-38
10.5.3 Debugging an Exit Handler 10-39

INDEX

Preface

Intended Audience

This manual is intended for experienced programmers who are working in the VAX/VMS environment. Although the examples are written in FORTRAN, you can use this manual as an aid to programming in other languages if you know the other language well and are able to read FORTRAN programs.

Structure of This Document

The *Guide to Programming on VAX/VMS* is designed to help programmers understand and use the features offered by the VAX/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 VAX/VMS operating system features to accomplish those tasks.

Ten chapters are included:

- Chapter 1 describes the communication between the program units that make up an executable program.
- Chapter 2 describes the makeup of the individual program units.
- Chapter 3 discusses process creation and control.
- Chapter 4 explains the mechanics of processing your program units into executable programs.
- Chapter 5 explains how to use the VAX/VMS Debugger to check your program for errors while it executes.
- Chapter 6 describes program data structures.
- Chapter 7 shows how to define a DCL command that invokes your program.
- Chapter 8 describes I/O techniques used when the typical input and output device is your terminal.

- Chapter 9 describes I/O techniques used to manipulate data stored in files.
- Chapter 10 explains how run-time errors in your program are handled.

Associated Documents

Complete descriptions of the VAX/VMS components discussed in this manual are provided in the appropriate reference volumes in the VAX/VMS document set. In particular, see the following volumes:

- *VAX/VMS DCL Dictionary*
- *VAX/VMS System Routines Reference Volume*
- *VAX/VMS Utility Routines Reference Manual*
- *VAX/VMS Run-Time Library Routines Reference Manual*

Specific information in other languages, can be found in the appropriate VAX programming manual for that language. For example, for a complete description of the VAX FORTRAN language, see *Programming in VAX FORTRAN*. Information on the VAX BASIC programming language can be found in *Programming in VAX BASIC*.

Conventions Used in This Document

Convention	Meaning
[RET]	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, [RET] .
[CTRL/x]	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.

Convention	Meaning
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square 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.

New and Changed Features

The focus of the *Guide to Programming on VAX/VMS* has changed to supply programmers working in any programming language with guidelines for using the development tools available with VAX/VMS. However, the manual still has a FORTRAN orientation; all examples are in FORTRAN, and the programmer is assumed to have a reading knowledge of the language.

The Debugger section contains most of the new and changed features for Version 4.4.

Using the Debugger (Changes in Chapter 5)

Several new debugger screen features have been added. Changes include new display window definitions, the addition of a PROMPT predefined display, greater usable screen height, new MOVE, EXPAND, and EXTRACT commands, and some new key pad definitions.

Because windows can now be divided vertically, some key definitions affecting screen displays have been changed. Key 7 now displays SRC in the top left half of the screen, INST in the top right half, OUT below these two, and the new PROMPT display under the OUT display. GOLD-7 now displays INST in the top left half, REG in the top right half, OUT below these two, and PROMPT under the OUT display. The BLUE-7 key is now undefined. For more detailed information about these and other new features, see the *VAX/VMS Debugger Reference Manual*.

1 Interunit Logic

An executable program (also called an executable image, applications system, or subsystem) consists of program units that are compiled and linked together (see Chapter 4). One program unit is the main program and the rest are subprograms. You invoke an executable program from DCL command level with either its command name (see Chapter 7) or the RUN command.

When designing a program, you decide what operations are required. Typically, in the initial stages of implementation, each program unit performs one of these operations. As you proceed with the detailed coding of the program, you create additional subprograms to factor out repetitive and complex operations. You can write each program unit as a separate file or group related subprograms into a single file (see Section 4.1).

When you invoke an executable program, program execution begins at the first executable statement in the main program unit. In FORTRAN, the main program unit begins with a PROGRAM statement (optional, but recommended) and ends with an END statement; it can invoke subprograms, which can then invoke other subprograms. In FORTRAN the following types of subprograms can be used (see your programming documentation for the types of subprograms that may be declared in other languages):

- Subroutine—A program unit beginning with a SUBROUTINE statement and ending with an END statement. To invoke a subroutine from another program unit, use a CALL statement.
- Function—A program unit beginning with a FUNCTION statement and ending with an END statement. A function provides the invoking program unit with a value of a particular data type. To invoke a function, use a function reference wherever a value of the same data type is acceptable.

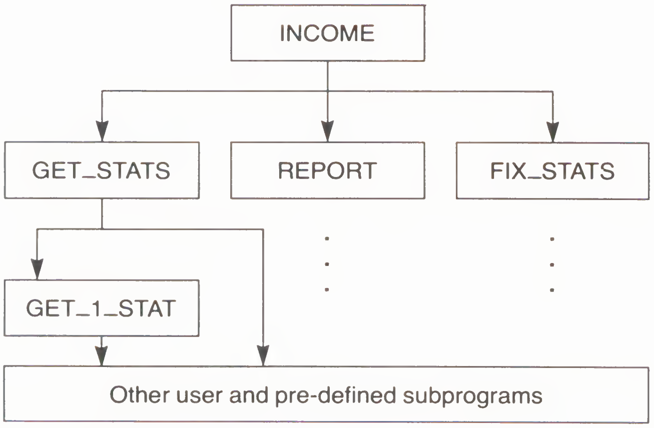
For clean, efficient code, use the prewritten system-defined subprograms and procedures defined specifically for your language whenever possible. System-defined procedures (also called system routines) can be called from any VAX language providing that language supports the data structure required by the particular routine. The following types of system-defined subprograms are allowed in FORTRAN:

- Run-Time Library and Utility routines—System-defined subprograms that perform common system and utility operations, such as terminal I/O (see the *VAX/VMS Run-Time Library Routines Reference Manual* and the *VAX/VMS Utility Routines Reference Manual*).

- System services—System-defined subprograms that perform system operations, such as defining error handlers (see Section 1.1.4 and the *VAX/VMS System Services Reference Manual*).
- Intrinsic subprograms—FORTRAN-defined subprograms that perform arithmetic, data type conversions, and character and bit manipulations (see Section 1.1.3).

The Run-Time Library procedures, unlike the system service procedures, usually use the FORTRAN default passing mechanisms. Therefore, when a Run-Time Library procedure and a system service procedure perform the same operation, the Run-Time Library procedure is generally preferred.

The following diagram shows a typical division of labor within a program. The FORTRAN sample program INCOME reads statistics (GET_STATS), compiles statistics (REPORT), and corrects statistics (FIX_STATS). Complex operations in each of these top-level program units are factored out into subprograms (GET_1_STAT). Some program units may use system-defined procedures to perform utility functions, such as terminal I/O.

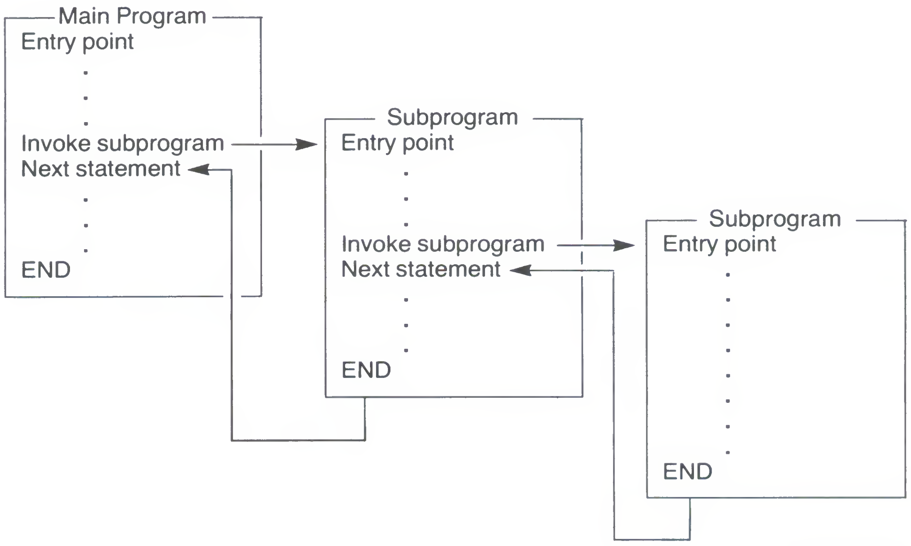


ZK-2022-84

1.1 Invoking Program Units

When you invoke an executable program from DCL command level, execution begins with the first executable statement—the entry point—of the main program. The main program's END statement terminates program execution, unless the program ended prematurely due to a STOP, a fatal or severe noncontinuable error, or another exception. If a program unit (the main program or a subprogram) invokes a subprogram, program control is transferred to the first executable statement of the specified subprogram. When the subprogram reaches its END statement, the subprogram terminates and program control returns to the invoking program unit at the statement following the subprogram invocation. The following figure diagrams program execution.

DCL command level is command level 0. Each time a program unit is invoked, control drops one level and the command level is incremented by one; when a program unit terminates, control returns one level and the command level is decremented by one. Thus, the main program in the following figure executes at command level one, the subprogram invoked by the main program at command level two, and the subprogram invoked by the first subprogram at command level three.



ZK-2023-84

1.1.1 Subroutine

To invoke a subroutine in FORTRAN, use a CALL statement. You can pass arguments to the subprogram by using an argument list or a common block (Section 1.2 contains more detail). The following statement invokes the subprogram GET_1_STAT using an argument list to pass three arguments.

```
CALL GET_1_STAT (LINE_NO,
2                COL_PERSONS,
2                PERSONS)
```

The following FORTRAN program unit is the subroutine GET_1_STAT invoked in the previous example.

```
SUBROUTINE GET_1_STAT (LINE_NO,   ! In
2                      COLUMN_NO, ! In
2                      STAT)      ! Out
              .
              .
              .
END  ! Subroutine
```

1.1.2 Function

A FORTRAN procedure must be called as a function if it returns a function value, or if it returns a condition value. A function provides the invoking program unit with the value of a particular data type. To invoke a function, use a function reference wherever you could use a value of the same data type. A function reference consists of the function name followed by an argument list. (The argument list allows FORTRAN to differentiate between a function reference and a variable, therefore the argument list must be specified even when no arguments are passed.) You can pass arguments to the function by using the argument list or a common block (Section 1.2 contains more details).

During execution a function assigns a value to the function name; this is the value provided to the invoking program unit. The data type of the function name, which determines the data type of the function value, must be specified in the function subprogram and the invoking program unit (the data type must be the same in both places). In the function subprogram, specify the data type as the first entity of the FUNCTION statement. In the invoking program unit, specify the data type of the function by including the function name in a type declaration statement. (If the invoking program unit does not specify the function name in a type declaration statement, implicit data typing applies; see Section 2.1.)

The following example shows the function reference used to invoke the GET_QUADAREA function. The function value provided to the invoking program by GET_QUADAREA is the area of the quadrangle (the value AREA, which is assigned to the function name, contains the area of the quadrangle).

```
REAL GET_QUADAREA,
2    QUADAREA,
2    SIDES (4),
2    ANGLES (4)
QUADAREA = GET_QUADAREA (SIDES,
2                        ANGLES)
```

The following is the GET_QUADAREA function.

```
REAL FUNCTION GET_QUADAREA (SIDES,  ! Passed
2                           ANGLES) ! Passed
! Dummy arguments
REAL SIDES(4),
2    ANGLES(4)
REAL AREA
! Calculate area
        .
        .
        .
! Assign AREA value to function name
GET_QUADAREA = AREA
END   ! Function
```

1.1.3 Intrinsic Subprograms

Intrinsic subprograms are FORTRAN-defined subprograms (most of which are functions) which perform arithmetic, data type conversions, and character and bit manipulations. If a user-written subprogram and an intrinsic subprogram have the same name, the user-written subprogram takes precedence (provided that the invoking program unit declares it in an EXTERNAL statement).

An intrinsic function does not have to be declared in a type statement. In addition, implicit data typing (default or that specified by an IMPLICIT statement) does not affect the data type of an intrinsic function.

Each intrinsic function may have a generic name, a specific name, or both.

- Specific—A specific name determines the data type of the subprogram's arguments and function value (if any).
- Generic—A generic name allows the data type of the arguments to determine which specific function should be executed.

Typically, when you invoke an intrinsic function, you use the generic name. If you specify an intrinsic function as an argument, however, you must use the specific name. (For details about generic names, see *Programming in VAX FORTRAN*).

In the following example, the specific function JIABS (generic name ABS) is executed twice. In the first call, since JIABS is executed, the argument must be an INTEGER*4 value. In the second call, since the argument is an INTEGER*4 value, JIABS is executed.

```
INTEGER*4 NUM,
2         RESULT
          .
          .
          .
RESULT = JIABS (NUM)
RESULT = ABS (NUM)
```

1.1.4 System-Defined Procedures

System-defined subprograms include Run-Time Library procedures, system service procedures, and utility procedures; for details see the *VAX/VMS Run-Time Library Routines Reference Manual*, the *VAX/VMS System Services Reference Manual*, and the *VAX/VMS Utility Routines Reference Manual*, respectively. Each of these manuals lists the procedures in alphabetical order, giving a detailed description of each, including arguments and return codes.

Note

Generally in the VAX/VMS document set, system-defined procedures are known as system routines. This manual refers to VAX/VMS system routines as procedures, because that is the term that most FORTRAN programmers are accustomed to.

In general, the system-defined procedures are named by a facility abbreviation, followed by a dollar sign and a descriptive name.

LIB$ERASE_LINE	Erases a line on the screen or in a buffer
SYS$SETEF	Sets an event flag

The following table lists commonly used facility abbreviations.

CLI	Command language editor procedures
DCX	Data compression procedures
EDT	Editor procedure
FDL	File definition procedures
LIB	General utility procedures
LBR	Library procedures
MTH	Math procedures
SMG	Screen management procedures
SOR	Sort and merge procedures
SYS	System service procedures

Most system-defined procedures are written as functions where the function value is the return status of the procedure (an INTEGER*4 value indicating success or failure of the routine). To access the return status, you must specify the procedure name as an INTEGER*4 data type and invoke the procedure as a function. If you invoke a system-defined procedure with a CALL statement, the procedure executes; however, you cannot examine the return status to determine whether it executed successfully.

The following statement invokes the Run-Time Library procedure LIB$GET_INPUT to read a line of input from the terminal screen.

```
INTEGER*4     STATUS,
2             LIB$ERASE_PAGE
CHARACTER*255 STRING
INTEGER*4     STRING_LEN
STATUS = LIB$GET_INPUT (STRING,
2                       'Input a string: ',
2                       STRING_LEN)
```

The possible return codes for a procedure are listed at the end of the routine template for the procedure (for a complete description of the routine template, see the *Introduction to VAX/VMS System Routines*). In general, the return codes are system-defined global symbols named by the facility abbreviation (see the previous table), followed by a dollar sign, an underscore, and an abbreviation of the error message. (The system service return codes use the facility abbreviation SS rather than SYS.)

LIB$_INVARG	General utility: invalid argument
SS$_ACCVIO	System service: access violation

A few of the Run-Time Library procedures return a function value that is not a status code (for details, see the *Introduction to VAX/VMS System Routines*). For example, the function value of Run-Time Library procedure LIB$LP_LINES is the default number of lines on a line printer page.

A Run-Time Library procedure whose function value is not a return status must be invoked as a function.

```
INTEGER*4 LENGTH,
2         LIB$LP_LINES
LENGTH = LIB$LP_LINES ()
```

1.2 Transfer of Data

Typically, to pass data between an invoking program unit and a subprogram you use an argument list (in FORTRAN, you can also use a common block).

1.2.1 Argument Lists

An argument list, which consists of one or more variable names or values separated by commas and delimited by parentheses, must be declared in two places:

- Subprogram invocation statement—An argument list in a subprogram invocation statement passes actual arguments to a subprogram. The arguments in this list are known as actual arguments.
- Subprogram definition statement—An argument list in a subprogram definition statement names the dummy arguments used in the subprogram. The arguments in this list are known as dummy arguments.

When you invoke a subprogram, the actual arguments are associated with the dummy arguments on a one-to-one basis. The first actual argument is associated with the first dummy argument, the second actual argument with the second dummy argument, and so on. Usually, when the subprogram exits, each actual argument has the value of its corresponding dummy argument. Since the actual arguments and dummy arguments are associated on a one-to-one basis, they must agree in number, order, and data type.

- Actual arguments—Actual arguments can be variables, constants, expressions, or subprogram names. Any actual argument whose corresponding dummy argument is modified by the subprogram must be a variable.

  If you use a subprogram name as an actual argument, the subprogram must be declared in an EXTERNAL or an INTRINSIC statement (as in FORTRAN) to identify the subprogram name as a global symbol defined in a different program unit. User- and system-defined subprograms are declared in EXTERNAL statements; FORTRAN intrinsic subprograms are declared in INTRINSIC statements. When using a FORTRAN intrinsic function as an actual argument, you must use the specific name rather than the generic name. (Typically, a subprogram does not have to be declared EXTERNAL or INTRINSIC because FORTRAN assumes that a name used in a CALL statement or function reference is a global symbol.)
- Dummy arguments—Dummy arguments are always variables. For clarity, a dummy argument should have a name that reflects its contents. (Mechanically, the names of actual and dummy arguments do not matter.)

In the FORTRAN example below the subprogram GET_1_STAT has three dummy arguments: LINE_NO, COLUMN_NO, and STAT. GET_1_STAT uses LINE_NO and COLUMN_NO to position the cursor on the screen, reads a value at that position, and assigns the value to STAT. When GET_STATS invokes GET_1_STAT, it passes three actual arguments: LINE_NO, COL_PERSONS, and PERSONS. The first two actual arguments can be variables, expressions, or constants as long as each argument evaluates to an integer. Since the third actual argument is modified by GET_1_STAT it must be a variable.

GETSTATS.FOR

```
               .
               .
               .
INTEGER*4 LINE_NO,
2         COL_PERSONS,
2         PERSONS
PARAMETER (COL_PERSONS = 12)
INTEGER*4 STATUS,
2         GET_1_STAT
EXTERNAL GET_1_STAT
               .
               .
               .
STATUS = GET_1_STAT (LINE_NO,     ! Actual 1
2                    COL_PERSONS, ! Actual 2
2                    PERSONS)     ! Actual 3
               .
               .
               .
```

GET1STAT.FOR

```
INTEGER FUNCTION GET_1_STAT (LINE_NO,   ! Dummy 1
2                            COLUMN_NO, ! Dummy 2
2                            STAT)      ! Dummy 3
! Declare dummy arguments
INTEGER*4 LINE_NO,
2         COLUMN_NO,
2         STAT
INTEGER*4 STATUS
               .
               .
               .
GET_1_STAT = STATUS
END
```

Each time you invoke a subprogram, check the argument lists carefully. If the actual arguments and the dummy arguments do not agree in number, order, data type, or structure, the subprogram receives incorrect data. A subprogram that receives incorrect data may execute without errors but return incorrect results, or it may generate an error. (In the early stages of program development, an access violation error may indicate disagreement between actual and dummy arguments.)

1.2.2 Common Blocks

In FORTRAN, a COMMON statement associates a list of variables with a contiguous area in memory called a common block. A program can have one unnamed common block and 250 named common blocks. When different program units refer to the same common block, they refer to the same area of memory. The first program unit to refer to a common block defines it.

If the common block is a named common block, place the name in slashes following the keyword COMMON. A common block name is a global symbol that must be unique among global symbols used by the program (subprogram names, other common block names, and so on). The following FORTRAN statement defines a common block of four bytes named UIC.

```
INTEGER*2 MEMBER,
2         GROUP
COMMON /UIC/ MEMBER,
2            GROUP
```

A variable should be specified in a data type declaration statement before being used in a COMMON statement; otherwise, FORTRAN assigns the variable an implicit data type. Since any program unit that references a common block can modify the data, constants (including those defined in PARAMETER statements) cannot be placed in a common block. In addition, subprogram names cannot be placed in a common block.

Variables are allocated space in a common block (and so in memory) according to their order in the COMMON block declaration. For example, in the previous program segment, the INTEGER*2 variable MEMBER refers to the low-order two bytes of the common block, the INTEGER*2 variable GROUP refers to the high-order two bytes. A different program unit can reference the data in the UIC common block as four bytes, two words (as in the previous example), or one longword. Generally, however, program units that reference the same common block should use variables that agree in number, order, and data type.

1.2.2.1 Primary Uses

Typically, you use common blocks to share data between program units when argument lists are unavailable or to share data between processes.

- Sharing data between program units—In cases where the VAX/VMS system invokes a subprogram that you have written, the subprogram is frequently invoked with no arguments or only one argument. To pass additional data to your subprogram, use a common block. (By using common blocks two subprograms can share data without either subprogram having to invoke or be invoked by the other. Such transfers of data make program maintenance difficult; if they cannot be avoided, they should be very well documented.)

- Sharing data between processes—A common block used to share data between processes must be installed as a shared shareable image; see Section 3.4.3.

In the FORTRAN example below COUNTAL.FOR counts the alias names associated with a library module by invoking the Run-Time Library procedure LBR$SEARCH. LBR$SEARCH invokes the user-written subprogram SEARCH to increment a counter each time it finds an alias name.

When LBR$SEARCH invokes a user-written subprogram, it passes two actual arguments, both of which contain values determined by LBR$SEARCH. To pass your own data to the user-written subprogram, you must use a common block. In this example, the common block SEARCH_VAR is used to pass a counter between COUNTAL.FOR and the user-written subprogram SEARCH.

COUNTAL.FOR

```
! Declare status variable
INTEGER STATUS

! Declare library index
INTEGER INDEX,
2       GLOBAL_INDEX
PARAMETER (GLOBAL_INDEX = 2)

! Declare module arguments
CHARACTER*31 MODNAME       ! Name of module
INTEGER MODNAME_LEN,       ! Length of module name
2       TXTRFA (2),        ! RFA for module text
2       COUNT              ! Counts alias names
! Declare common block
COMMON /SEARCH_VAR/ COUNT  ! The third variable
                           ! passed to SEARCH
                    .
                    .
                    .
! Search procedure
EXTERNAL SEARCH
INTEGER SEARCH
```

```
! Begin code
                       .
                       .
                       .
! Search for alias names in index number 2
STATUS = LBR$SEARCH (INDEX,          ! Library index
2                    GLOBAL_INDEX,   ! Primary index
2                    TXTRFA,         ! RFA of key
2                    SEARCH)         ! User procedure
TYPE *,'Count of alias names: ', COUNT
END  ! Subprogram
```

SEARCH.FOR

```
INTEGER FUNCTION SEARCH (ALIASNAME,
2                        RFA)
! Function called by LBR$SEARCH for each alias name
! counts the alias names
! LBR$SEARCH defined arguments
CHARACTER*(*) ALIASNAME   ! Name of module
INTEGER RFA (2)           ! RFA of module
! User-defined argument
COMMON /SEARCH_VAR/ COUNT
INTEGER STATUS_OK
PARAMETER (STATUS_OK = 1)
COUNT = COUNT + 1         ! Update alias count
SEARCH = STATUS_OK        ! Return good status to LBR$SEARCH
END   ! Function
```

1.2.2.2 Initialization

When a FORTRAN common block must be initialized, best practice is to use a BLOCK DATA subprogram. Any program unit that references a common block can initialize the data in that common block; however, if more than one program unit attempts to initialize the same common block the results are unpredictable, depending on the order in which the program units are linked. Mechanically, a common block initialized by a BLOCK DATA subprogram is no different from a common block initialized in any other program unit. However, by initializing all common blocks in BLOCK DATA subprograms, you ensure that no two program units initialize the same common block.

A BLOCK DATA subprogram is a program unit beginning with a BLOCK DATA statement and ending with an END statement. BLOCK DATA subprograms may contain type declaration, PARAMETER, COMMON, EQUIVALENCE, DATA, IMPLICIT, DIMENSION, SAVE, and RECORD statements; they may not contain executable statements.

The following BLOCK DATA subprogram initializes a common block named UIC. To access the UIC common block from another program unit, declare a common block of the same name.

```
BLOCK DATA INIT_UIC
! Initializes the UIC common block with UIC [210,200]
INTEGER*2    MEMBER /'200'O/,
2            GROUP /'210'O/
COMMON /UIC/ MEMBER,
2            GROUP
END
```

1.3 Passing Control Information Among User Subprograms

Control information is information passed between program units to determine internal processing, such as the sequence in which statements are to be executed (execution sequence). In FORTRAN, flags, masks, and indicative values can be used to pass control information between program units.

Control variables used by more than one program unit should be treated on a global basis (see Section 4.2.4).

1.3.1 Flags

A flag indicates one of two conditions. Use a variable of LOGICAL data type to represent a flag. The following program segment uses the READ_ONLY flag to determine which OPEN statement to execute.

```
SUBROUTINE OPEN_FILE (FILENAME,
2                     READ_ONLY)
! Declare dummy arguments
CHARACTER*(*) FILENAME
LOGICAL READ_ONLY
! Logical unit number for chapter file
INTEGER CHAP_LUN
IF (READ_ONLY)  THEN
   OPEN (UNIT   = CHAP_LUN,
2        FILE   = FILENAME,
2        READONLY,
2        STATUS = 'OLD')
ELSE
   OPEN (UNIT   = CHAP_LUN,
2        FILE   = FILENAME,
2        STATUS = 'OLD')
END IF
END
```

1.3.2 Masks

A mask indicates one or more of a number of conditions. Use a longword for the mask. Each bit in the longword represents a particular condition; if a condition is met, the appropriate bit is set. For clarity, use the FORTRAN PARAMETER statement to assign meaningful symbolic names to the bit offsets.

In the following program segment, the QUALIFIER mask indicates which qualifiers were specified on the command line. The conditional statement ensures that the selected qualifiers are not mutually exclusive.

```
LOGICAL BTEST
! Mask
INTEGER*4 QUALIFIER
! Offsets
INTEGER*4 FORWARD,
2         BACKWARD,
2         GENERAL,
2         EXACT,
2         BEGIN,
2         END
PARAMETER (FORWARD  = 1,
2          BACKWARD = 2,
2          GENERAL  = 3,
2          EXACT    = 4,
2          BEGIN    = 5,
2          END      = 6)
! Condition codes
EXTERNAL BAD_QUALIFIER
          .
          .
          .
! /BEGIN and /END are mutally exclusive
! /GENERAL and /EXACT are mutally exclusive
IF ((BTEST (QUALIFIER,BEGIN) .AND. BTEST (QUALIFIER,END)) .OR.
2   (BTEST (QUALIFIER,GENERAL) .AND. BTEST (QUALIFIER,EXACT)))
2  CALL LIB$SIGNAL (%VAL(%LOC(BAD_QUALIFIER)))
          .
          .
          .
```

1.3.3 Indicative Values

An indicative value indicates exactly one of a number of conditions. Use a variable of INTEGER data type to represent the indicative value and integer constants to represent the possible conditions. For clarity, use the FORTRAN PARAMETER statement to assign meaningful symbolic names to the integer constants. The following program segment uses the indicative value COMMAND to determine which program unit to execute.

```
                  .
                  .
                  .
! Command entered
INTEGER COMMAND
! Possible commands
INTEGER COMMAND_CHAPTER,
2       COMMAND_TC,
2       COMMAND_INDEX,
2       COMMAND_SEARCH
PARAMETER (COMMAND_CHAPTER = 1,
2          COMMAND_TC = 2,
2          COMMAND_INDEX = 3,
2          COMMAND_SEARCH = 4)
                       .
                       .
                       .
! Get a command
STATUS = GET_COMMAND (COMMAND)
! Perform commands until EXIT
DO WHILE (STATUS)
    IF (COMMAND .EQ. COMMAND_CHAPTER)
2   STATUS = PERUSE_CHAPTER (BOOK,
2                            CHAPTER_NO,
2                            SECTION_NO,
2                            BEGIN_TEXT,
2                            END_TEXT)
    IF (COMMAND .EQ. COMMAND_TC)
2   STATUS = PERUSE_TC (BOOK,
2                       CHAPTER_NO,
2                       SECTION_NO,
2                       BEGIN_TEXT,
2                       END_TEXT)
                       .
                       .
                       .
     ! Get another command
     IF (STATUS) STATUS = GET_COMMAND (COMMAND)
END DO
                       .
                       .
                       .
```

1.4 Passing Variable-Length Data

An array or character string passed as an actual argument must have a specified length. Dummy character strings and arrays may also be declared using a specified length. However, since you may want to specify different actual arguments each time you invoke a subprogram, passed-length character strings, adjustable arrays, and assumed-size arrays allow a subprogram to include arrays and character strings in type statements without specifying their lengths.

Data storage for dummy character strings and arrays is allocated when the dummy argument is given a value; the storage remains allocated for the duration of the program. To conserve storage, you may want to use the library routines LIB$GET_VM and LIB$FREE_VM (see Section 2.1.6) to allocate and deallocate the storage used by the subprogram. Allocation and deallocation of storage is also recommended for general utility routines since you cannot anticipate the storage requirements of the invoking program.

1.4.1 Character Strings

A dummy argument declared as a passed-length character string can be associated with any actual argument of data type CHARACTER. A subprogram declares a passed-length character string by specifying the length of the string as an asterisk enclosed in parentheses.

```
CHARACTER*(*) TITLE
```

The dummy CHARACTER argument assumes the length of the actual CHARACTER argument. To avoid unwanted spaces at the end of the character string in the subprogram, specify the actual argument as a substring.

In the following example, the invoking program unit passes two substrings to the subprogram REPORT.

```
                    .
                    .
                    .
CHARACTER*26 TITLE
CHARACTER*10 FIRM
INTEGER TITLE_LEN,
2       FIRM_LEN
INTEGER STATS (1020)
                    .
                    .
                    .
CALL REPORT (TITLE(1:TITLE_LEN),
2            FIRM(1:FIRM_LEN),
2            STATS)
                    .
                    .
                    .
```

The associated dummy arguments in REPORT are passed-length character strings. (Since the dummy arguments are associated with substrings, no substring specification is needed when REPORT passes the dummy argument TITLE to the Run-Time Library procedure LIB$PUT_OUTPUT.)

```
SUBROUTINE REPORT (TITLE,
2                  FIRM,
2                  STATS)
CHARACTER*(*) TITLE,
2             FIRM
INTEGER STATS (1020)
                  .
                  .
                  .
! Write header
STATUS = LIB$PUT_OUTPUT (TITLE)
                  .
                  .
                  .
```

1.4.2 Arrays

To pass an array as an actual argument, specify the array name. The first element of the actual array is associated with the first element of the dummy array, the second element of the actual array with the second element of the dummy array, and so on.

A dummy array argument can be associated with any actual array of the same data type. Unless you have a specific reason for modifying an array, the dimensions of the dummy array, with the possible exception of the last dimension, should be the same as the dimensions of the actual array. Because of the way arrays are stored (see Section 6.9.4), changing the dimensions of an array affects the location of the array elements.

Typically, you declare a dummy array as an adjustable or assumed-size array.

- Adjustable array—Specify a dummy argument as an adjustable array by using dummy arguments and common block variables of data type INTEGER to dimension the array. (Expressions can be used as dimensions, however, the expressions must consist only of dummy arguments, common block variables, and constants.) In the following example, the invoking program unit passes the two-dimensional array NAME_LIST and its two dimensions ROWS and COLUMNS to the subprogram SEARCH.

```
! Declare array and dimensions
CHARACTER*10 NAME_LIST (10,12)
INTEGER*4    ROWS /10/,
2            COLUMNS /12/
CHARACTER*10 NAME
INTEGER*4    NAME_LEN
                .
                .
                .
CALL SEARCH (NAME_LIST,
2            ROWS,
2            COLUMNS,
2            NAME,
2            NAME_LEN)
                .
                .
                .
```

  SEARCH declares the dummy array as an adjustable array.

```
SUBROUTINE SEARCH (ARRAY,
2                  ROWS,
2                  COLUMNS,
2                  KEYWORD,
2                  KEYWORD_LEN)
! Find KEYWORD in ARRAY
! Argument declarations
CHARACTER*10 ARRAY (ROWS,COLUMNS)
INTEGER*4    LENGTH
CHARACTER*10 KEYWORD
INTEGER*4    KEYWORD_LEN
                  .
                  .
                  .
END     ! Subroutine SEARCH
```

- Assumed-size array—Specify a dummy argument as an assumed-sized array by specifying the last dimension of the array as an asterisk. If the dummy array is multidimensional, all dimensions (with the exception of the last) should be the same as those of the actual array. In the following example, the invoking program unit passes the array NAME_LIST and the position of the last element in NAME_LIST to SEARCH.

```
! Declare array
CHARACTER*10 NAME_LIST (12)
INTEGER*4    LAST
CHARACTER*10 NAME,
INTEGER*4    NAME_LEN
                    .
                    .
                    .
! Invoke SEARCH
CALL SEARCH (NAME_LIST,
2            LAST,
2            NAME,
2            NAME_LEN)
                    .
                    .
                    .
```

SEARCH declares the dummy array as an assumed-size array and uses the position of the last element to avoid referencing an out-of-bounds array element.

```
SUBROUTINE SEARCH (ARRAY,
2                  LAST,
2                  KEYWORD,
2                  KEYWORD_LEN)
! Find KEYWORD in ARRAY
! Dummy arguments
CHARACTER*10 ARRAY (*)
INTEGER*4    LAST
CHARACTER*10 KEYWORD
INTEGER*4    KEYWORD_LEN
                    .
                    .
                    .
END     ! Subroutine SEARCH
```

Adjustable and assumed-size array notations are not mutually exclusive. In the following example, the subprogram SEARCH declares the dummy array using dummy arguments for the first two dimensions (adjustable array notation) and an asterisk for the third dimension (assumed-size notation).

```
SUBROUTINE SEARCH (ARRAY,
2                  ROWS,
2                  COLUMNS,
2                  KEYWORD,
2                  KEYWORD_LEN)
! Find KEYWORD in ARRAY
! Argument declarations
CHARACTER*10 ARRAY (ROWS,COLUMNS,*)
INTEGER*4    LENGTH
CHARACTER*10 KEYWORD
INTEGER*4    KEYWORD_LEN
                   .
                   .
                   .
END      ! Subroutine SEARCH
```

Declaring an assumed-sized array generally takes less overhead than declaring an adjustable array; therefore, assumed-sized arrays are often used for one-dimensional arrays. However, since a subprogram cannot compute the length of an assumed-size array, you must use adjustable arrays under the following circumstances:

- You want to reference the entire dummy array by including its name in an I/O list.
- You want to enable bounds checking for the dummy array (FORTRAN /CHECK=BOUNDS).

1.5 Passing Arguments to System-Defined Procedures

The VAX/VMS documentation for each system procedure describes the arguments that must be, or can be, passed to the procedure. It is recommended that you omit any optional argument that is unnecessary or that you want equated to its default value. Include commas to indicate that an argument has been omitted. If you are using a Run-Time Library or Utility procedure, you can omit trailing commas; if you are using a system service procedure, you cannot.

Note

When you omit an optional argument, FORTRAN uses the by-value passing mechanism to pass a value of 0 in place of the omitted argument . If the argument would normally have been passed by reference or passed by descriptor, the 0 value FORTRAN passes generally indicates to the system-defined procedure that it should use a default value (if one exists). If the argument would normally be passed by value, the system-defined procedure uses the 0 value rather than any existing default value. Other programming languages may not pass or receive arguments

in the same way as FORTRAN does. Therefore, check the default values for your programming language.

Each argument description in the *VAX/VMS System Routines Reference Volume* indicates the VMS usage, access method, data type, passing mechanism, and value of the argument. For an explanation of how this information is conveyed, see the *Introduction to VAX/VMS System Routines*. If the system-defined procedure reads data from the argument, the actual argument can be a variable, constant, or expression. If the system-defined procedure writes data to the argument, the actual argument must be a variable.

The procedure descriptions are not FORTRAN specific; therefore, the argument data types are indicated by size. The following table matches the size terminology used in the descriptions with the FORTRAN data types. If you pass an argument of the wrong type, no message reports the error; however, the procedure receives incorrect data and an error may occur when you execute the program.

Size	FORTRAN Data Type
Byte	BYTE, LOGICAL*1
Word	INTEGER*2, LOGICAL*2
Longword	INTEGER*4, LOGICAL*4, REAL*4
Quadword	REAL*8, COMPLEX*8
Character string	CHARACTER[1]
Routine name	subprogram name[2]
Variable-length data structure	RECORD

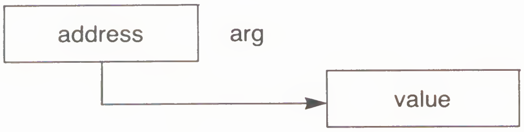
[1]Since many of the procedures do not allow trailing blanks, you should specify a substring.

[2]A subprogram name that is passed as an argument must be declared in an EXTERNAL or INTRINSIC statement.

1.5.1 Mechanics of Passing Arguments

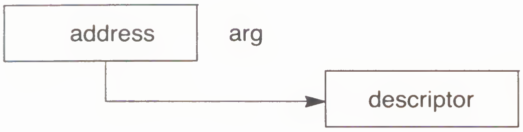
Arguments are passed in one of three ways:

- By reference—The address of the argument is passed in a longword.



ZK-2024-84

- By descriptor—The address of the argument's descriptor is passed in a longword. The format of the descriptor depends on the data type of the argument and may be two or more longwords.



ZK-2025-84

- By value—The value of the argument is passed in a longword; the value must be a 32-bit value.

ZK-2026-84

The following table lists the FORTRAN built-in functions that allow you to specify a particular passing mechanism. Typically, you use these functions only to override the default passing mechanisms.

Function	Passing mechanism
%REF	Passes the argument list entry by reference.
%DESCR	Passes the argument list entry by descriptor.
%VAL	Passes the argument list entry by value. If the actual argument is shorter than 32 bits, it is sign extended to a 32-bit field.[1]

[1]To sign extend a value, the high-order bits of the longer field are set to the sign bit of the shorter value.

The passing mechanisms used by the subprogram and the invoking program unit must agree. By default, FORTRAN passes logical and numeric data by reference and character data by descriptor. These default passing mechanisms always apply when you pass arguments between FORTRAN program units; however, other languages may not use these particular default passing mechanisms. When you pass arguments to system-defined procedures, some exceptions to the default passing mechanisms occur.

The most frequent exception to the default passing mechanisms occurs when a system-defined procedure requires that an argument be passed by value. For example, the system service procedure SYS$CLREF requires one argument; that argument must be passed by value.

```
INTEGER*4 EV_FLAG
! Declare status and system procedures
INTEGER*4 STATUS,
2         SYS$CLREF
STATUS = SYS$CLREF (%VAL(EV_FLAG))
```

1.5.2 Aligning Data

In rare cases, a system-defined procedure requires that an argument be aligned. That is, the data in memory must begin on a byte, word, longword, quadword, or page boundary (the routine description specifies what type of alignment is required). To align data:

1. Place the data in a common block.
2. Create an options file containing a PSECT_ATTR link option of the form

   ```
   PSECT_ATTR = name, alignment
   ```

   Specify **name** as the name of the common block that contains the data. Specify **alignment** as one of the following keywords: BYTE, WORD, LONG, QUAD, or PAGE.
3. Link the options file with your program.

Assuming that the program in SECTIONS.OBJ contains a common block named GLOBAL_SECTION, the following command page aligns the data in that common block.

```
$ LINK SECTIONS,PAGE_ALIGN.OPT/OPTION
```

The options file, PAGE_ALIGN.OPT, contains the PSECT_ATTR link option.

```
PSECT_ATTR = GLOBAL_SECTION, PAGE
```

1.5.3 Passing Bytes

To pass the address of a byte size argument to a system-defined procedure (that is, to pass a byte value by reference), declare the actual argument as a byte value. In the following example, LIB$DEC_OVER uses the value in the BYTE variable OVERFLOW to determine whether to enable or disable decimal overflow detection.

```
! Argument for LIB$DEC_OVER
BYTE OVERFLOW /1/
INTEGER OLD_OVERFLOW
OLD_OVERFLOW = LIB$DEC_OVER (OVERFLOW)
```

1.5.4 Passing Words

To pass the address of a word size argument to a system-defined procedure (that is, to pass a word value by reference), specify the actual argument as a 2-byte integer value. In the following example, LIB$GET_INPUT reads information from the terminal and returns the number of characters read in the INTEGER*2 variable AGE_LEN.

```
! Arguments for LIB$GET_INPUT
CHARACTER*3 AGE
INTEGER*2 AGE_LEN
! Declare status and system procedures
INTEGER*4 STATUS,
2         LIB$GET_INPUT
STATUS = LIB$GET_INPUT (AGE,           ! String read
2                       'Enter age: ', ! Prompt
2                       AGE_LEN)       ! Length of AGE
```

1.5.5 Passing Longwords

To pass the address of a longword size argument to a system-defined procedure (that is, to pass a longword value by reference), specify the actual argument as a 4-byte integer value. In the following example, SYS$SUSPND suspends the process whose process identification number is specified by the INTEGER*4 variable PID.

```
! Argument for SYS$SUSPND
INTEGER PID

! Declare status and system procedures
INTEGER*4 STATUS,
2         SYS$SUSPND

STATUS = SYS$SUSPND (PID,)
```

System-defined procedures may require that you specify values for bits, bytes, or words within a longword argument. The following subsections describe how to specify elements within a longword.

1.5.5.1 Specifying Byte and Word Values

To specify byte or word values within a longword, define the longword as a record (or an array). The elements of the longword must be specified in a record (or an array) to ensure that they are stored contiguously in memory. The first byte or word field is the low-order byte or word of the longword; the last byte or word field is the high-order byte or word of the longword. The following program segment passes a UIC of [210,200] to SYS$CREPRC. You cannot pass a RECORD by value; therefore, a UNION is used to create a longword field that can be passed by value.

```
STRUCTURE /UIC/
 UNION
  MAP
   INTEGER*2 MEMBER /'200'O/
   INTEGER*2 GROUP /'210'O/
  END MAP
  MAP
   INTEGER*4 NUMBER
  END MAP
 END UNION
END STRUCTURE

RECORD /UIC/ MY_UIC
INTEGER*4 STATUS,
2         SYS$CREPRC

STATUS = SYS$CREPRC (,
2                    'SYS$USER:[ACCOUNT]ADDEND',
2                    ,,,,,,
2                    %VAL(4),
2                    %VAL(MY_UIC.NUMBER),,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

END
```

1.5.5.2 Setting Bits

If a system-defined procedure requires that a longword contain a mask, certain bits in the longword carry special meaning for the procedure. To set bits in a mask, use one of the following techniques:

1. Equate the longword to a system-defined mask value
2. Set the appropriate bit(s) within the longword (the intrinsic function IBSET sets a bit)

Typically, the description of a mask argument lists the system-defined symbols associated with each bit offset. For example, the **flags** argument description of the SYS$CRMPSC system service includes the following table:

Flag	Description
SEC$M_GBL	Pages form a global section. Default is private section.
SEC$M_CRF	Pages are copy-on-reference. By default, pages are shared.
SEC$M_DZRO	Pages are demand-zero pages. By default, they are not.
.	.
.	.
.	.
SEC$M_WRT	Pages form a read/write section. By default, pages form a read-only section.

Generally, two sets of symbols are associated with each bit: a $V form that represents the bit offset and a $M form that defines a mask value for the bit offset. For example, to reference the bit offset for the **flags** argument of SYS$CRMPSC, use the prefix SEC$V rather than SEC$M. The symbols for bit offsets are defined in the same module as the symbols for mask values.

To create pages for read and write using the $M mask value, use SEC$M_WRT as the mask value; all bits except SEC$V_WRT are set to 0. (The SEC$ symbols are defined in module $SECDEF of the system object or shareable image library, and also of the FORTRAN system definition library; see Sections 4.2.4 and 4.2.5.)

```
INCLUDE '($SECDEF)'
STATUS = SYS$CRMPSC (ADDR,
2                    RET_ADDR,
2                    ,
2                    %VAL(SEC$M_WRT),  ! Mask
2                    ,,,
2                    %VAL(CHAN),
2                    ,,,)
```

To create copy-on-reference pages for read and write access, use the logical .OR. operator to combine the values of SEC$M_CRF and SEC$M_WRT; all bits except SEC$V_CRF and SEC$V_WRT are set to 0. (The MASK variable in the following example is used for clarity; you could pass the logical expression.)

```
INCLUDE '($SECDEF)'
INTEGER*4 MASK /0/
MASK = SEC$M_CRF .OR. SEC$M_WRT
STATUS = SYS$CRMPSC (ADDR,
2                    RET_ADDR,
2                    ,
2                    %VAL(MASK),  ! Mask
2                    ,,,
2                    %VAL(CHAN),
2                    ,,,)
```

To use the $M mask values without zeroing all other bits in the mask, use the logical .OR. operator to combine the current mask value with the new mask value.

```
INCLUDE '($SECDEF)'
INTEGER MASK
MASK = SEC$M_WRT
       .
       .
       .
MASK = MASK .OR. SEC$M_CRF
```

To create copy-on-reference pages for read and write access using the $V bit offsets, set the SEC$V_CRF and the SEC$V_WRT bits as shown in the following example.

```
INCLUDE '($SECDEF)'
INTEGER MASK /0/
MASK = IBSET (MASK,SEC$V_CRF)
MASK = IBSET (MASK,SEC$V_WRT)
STATUS = SYS$CRMPSC (ADDR,
2                    RET_ADDR,
2                    ,
2                    %VAL(MASK),  ! Mask
2                    ,,,
2                    %VAL(CHAN),
2                    ,,,)
```

1.5.6 Passing Quadwords

To pass the address of a quadword size argument to a system-defined procedure (that is, to pass a quadword value by reference), specify either a two-element INTEGER*4 array or a record. The elements of a quadword must be specified in an array or record to ensure that they are stored contiguously in memory. In the following example, SYS$BINTIM converts the specified absolute time into binary time and returns the binary time as a quadword value.

```
! Absolute time
CHARACTER*(*) TIME_STR
PARAMETER (TIME_STR = '23-JAN-1983  14:30:22.00')
! Binary time
INTEGER*4 TIME(2)
! Declare system procedures
INTEGER*4 SYS$BINTIM
STATUS = SYS$BINTIM (TIME_STR,
2                    TIME)
```

System-defined procedures may require that you specify values for bits, bytes, words, or longwords within a quadword argument. The following subsections describe how to specify elements within a quadword.

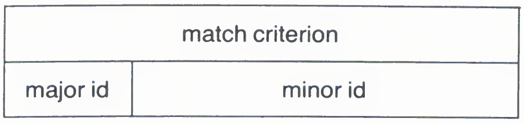
1.5.6.1 Specifying Byte, Word, and Longword Values

To specify byte, word, or longword values within a quadword, define the quadword as a record. The elements of a quadword must be specified in a record (or an array) to ensure that they are stored contiguously in memory. The first byte, word, or longword field is the low-order byte, word, or longword of the quadword; the last byte, word, or longword field is the high-order byte, word, or longword of the quadword.

The **iosb** argument of SYS$GETJPI requires a quadword containing two words followed by a longword: the low-order word contains the final status, the second word is undefined, and the longword contains a value of 0. (%FILL is a pseudofield name defined by FORTRAN; see Section 6.10.1.1.)

```
! Status variable and routine
INTEGER*4 STATUS,
2         SYS$GETJPI
! I/O status block for $GETJPI
STRUCTURE /IOSB/
 INTEGER*2 STATUS,
2          %FILL
 INTEGER*4 ZERO /0/
END STRUCTURE
RECORD /IOSB/ JPI_IOSB
                     .
                     .
                     .
STATUS = SYS$GETJPI (,,,
2                    JPI_LIST,    ! List
2                    JPI_IOSB,,)  ! Status
                     .
                     .
                     .
```

In another case, the **ident** argument of the SYS$CRMPSC system service requires a quadword in the format:



ZK-2027-84

Specify the quadword as a record containing a longword for the match criterion, followed by a union containing two MAP structures: a longword for the minor ID and a four-element BYTE array for the major ID. Assign the match criterion to the first field, CRITERION. (Match criteria values are defined by the global symbols SECK_MATALL, SECK_MATEQU, and SEC$K_MATLEQ, that are defined in $SECDEF.) Assign the minor ID value to the first MAP structure of the second field, MINOR_ID. Assign the major ID value to the fourth element of the BYTE array, MAJOR_ID (4), in the second MAP structure of the second field. (The minor ID value must be assigned first; otherwise, the high-order byte of **minor id** value will overwrite **major id**.)

```
! Declare ident structure
STRUCTURE /MATCH/
  INTEGER*4 CRITERION
  UNION
   MAP
    INTEGER*4 MINOR_ID
   END MAP
   MAP
    BYTE MAJOR_ID(4)
   END MAP
  END UNION
END STRUCTURE
! Define ident record
RECORD /MATCH/ DATA_ID
! Define symbols for match criteria
INCLUDE '($SECDEF)'
! Status variable and routine
INTEGER*4 STATUS,
2         SYS$CRMPSC
           .
           .
           .
! Assign values to ident record
DATA_ID.CRITERION = SEC$K_MATEQU
DATA_ID.MINOR_ID = 100
DATA_ID.MAJOR_ID(4) = 1
! Map section
STATUS = SYS$CRMPSC (IN_ADDR,
2                    OUT_ADDR,
2                    ,
2                    %VAL(SEC$M_GBL .OR. SEC$M_WRT),
2                    'GLOBAL_SEC',
2                    DATA_ID,
2                    ,
2                    %VAL(CHAN),
2                    ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
           .
           .
           .
```

1.5.6.2 Setting Bits

To specify a quadword mask, define the quadword as a two-element INTEGER*4 array. Bit positions for a quadword mask range from 0 to 63: bit positions 0 through 31 are in the low-order longword; positions 32 through 63 are in the high-order longword. Use IBSET to set a bit in either longword; see Section 1.5.5.2.

The following program segment grants the user DETACH and SYSNAM privileges for the duration of image execution. Specifying the first argument of SYS$SETPRV as 1 enables the privileges specified by PRIV_MASK. Dividing the bit position by 32 determines whether the bit should be set in the low-order or high-order longword; a value of 0 indicates the low-order longword. (Note that the array subscripts are 0 and 1.) The remainder resulting from the division operation indicates the bit offset. (The global symbols that define the bit offsets are defined in the $PRVDEF module of the FORTRAN system definition library; see Section 4.2.5.)

```
! Include the privilege symbol definitions
INCLUDE '($PRVDEF)'
! Privilege mask
INTEGER*4 PRIV_MASK (0:1),
2         ISUB,
2         IPOS
! Intrinsic function
INTRINSIC MOD
! Status variable and system services
INTEGER*4 STATUS,
2         SYS$SETPRV
                      .
                      .
                      .
ISUB = PRV$V_DETACH/32
IPOS = MOD (PRV$V_DETACH,32)
PRIV_MASK (ISUB) = IBSET (PRIV_MASK(ISUB), IPOS)
ISUB = PRV$V_SYSNAM/32
IPOS = MOD (PRV$V_SYSNAM,32)
PRIV_MASK (ISUB) = IBSET (PRIV_MASK(ISUB), IPOS)
STATUS = SYS$SETPRV (%VAL(1),     ! Enable privs
2                     PRIV_MASK,   ! Privs
2                     ,,)
                      .
                      .
                      .
```

1.5.7 Passing Variable-Length Data Structures

To pass a variable-length data structure to a system-defined procedure, specify a record. The VAX/VMS system routines documentation generally describes variable-length data structures with a diagram of the structure, followed by a description of each element in the structure. The elements are described in order from right to left, beginning with the first longword of the structure. This is the order in which the elements must be stored in memory and, consequently, the order of the fields in the record.

To pass a variable-length data structure for the **itmlst** argument to SYS$GETJPI:

1 Define **itmlst** record structure—Each record contains either the four fields pictured in the **itmlst** description or a single longword to end the list of item codes. The following program segment defines a record structure that alternately contains four fields or one field.

```
! 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
```

A UNION block within a STRUCTURE block may contain one or more MAP blocks. When the fields within one MAP block are defined, the fields within other MAP blocks are undefined. (Section 6.10 discusses records.)

2 Declare **itmlst** variable—Decide what information you require. Declare a RECORD array containing one element for each item code, plus an extra element for the end-of-list item. The following statement declares an ITMLST RECORD array for two item codes.

```
RECORD /ITMLST/ JPI_LIST (3)
```

3 Declare information buffers—For each item code, define two variables as buffers: a buffer to contain the information returned by SYS$GETJPI and a buffer to contain the length in bytes of the returned information. The following program segment declares buffers for the item codes JPI$_PRCNAM (process name) and JPI$_PRI (process priority).

```
! Declare buffers for information
CHARACTER*15 NAME
INTEGER*4    PRIORITY
INTEGER*2    NAME_LEN,
2            PRIORITY_LEN
```

4 Assign field values—For each item code, assign values to the appropriate fields of an array element as described in following paragraphs. The following examples assign values to the first element of JPI_LIST, which is for the JPI$_PRCNAM item code. (You would assign values to the second element of JPI_LIST in the same way as for the JPI$_PRI item code.)

Assign the BUFLEN field the length in bytes of the buffer for the requested information.

```
JPI_LIST(1).BUFLEN = 15
```

Assign the CODE field the value of the item code. (The request codes for SYS$GETJPI are defined in the $JPIDEF module of the system object or shareable image library, and also of the FORTRAN system definition library; see Sections 4.2.4 and 4.2.5.)

```
! Definitions of SYS$GETJPI item codes
INCLUDE '($JPIDEF)'
JPI_LIST(1).CODE = JPI$_PRCNAM
```

Assign the BUFADR field the address of the buffer for the requested information. (The %LOC built-in function returns the address of the specified variable.)

```
JPI_LIST(1).BUFADR = %LOC(NAME)
```

Assign the RETLENADR field the address of the buffer for the length of the requested information. (The %LOC built-in function returns the address of the specified variable.)

```
JPI_LIST(1).RETLENADR = %LOC(NAME_LEN)
```

5 End the list of item codes—Assign the END_LIST field of the last array element a value of 0.

```
JPI_LIST(3).END_LIST = 0
```

The following program invokes SYS$GETJPI to print the name (JPI$_PRCNAM) and priority (JPI$_PRI) of the current process.

```
PROGRAM GETJPI
! Displays the name and priority of the current process
! 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    NAME
INTEGER*4       PRIORITY
INTEGER*4       NAME_LEN,
2               PRIORITY_LEN
! Declare status and routine
INTEGER*4       STATUS,
2               SYS$GETJPI
! Set up itmlst
JPI_LIST(1).BUFLEN =    15
JPI_LIST(1).CODE =      JPI$_PRCNAM
JPI_LIST(1).BUFADR =    %LOC(NAME)
JPI_LIST(1).RETLENADR = %LOC(NAME_LEN)
JPI_LIST(2).BUFLEN =    4
JPI_LIST(2).CODE =      JPI$_PRI
JPI_LIST(2).BUFADR =    %LOC(PRIORITY)
JPI_LIST(2).RETLENADR = %LOC(PRIORITY_LEN)
JPI_LIST(3).END_LIST =  0
! Call SYS$GETJPI
STATUS = SYS$GETJPI (,,,JPI_LIST,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Display information
TYPE *,'NAME: ',NAME(1:NAME_LEN)
TYPE *,'PRIORITY: ',PRIORITY
END
```

1.6 Reading Information Returned from System-Defined Procedures

Typically, to use information returned by a system-defined procedure, you reference the actual argument that contains the information after the system-defined procedure executes. However, if the procedure returns the information as a mask, address, or a buffer containing multiple pieces of information, you must extract the data from the actual argument before it is useful.

1.6.1 Masks

To extract information from a mask, use the intrinsic function BTEST to test whether or not a particular bit is set. The following program segment uses the SYS$READEF system service to read local event flag cluster 1, and then determines whether flags 2, 3, and 4 are set. Since flag 2 is passed to the system service, the return status is used to determine whether that flag is set.

```
INCLUDE '($SSDEF)'  ! Defines SS$_ return codes
INTEGER*4 FLAG2,
2         FLAG3,
2         FLAG4,
2         CLUSTER
PARAMETER (FLAG2 = 2,
2          FLAG3 = 3,
2          FLAG4 = 4)
! Examine local event flag cluster
STATUS = SYS$READEF (FLAG2,
2                    CLUSTER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (STATUS .EQ. SS$_WASSET) THEN
               .
               . ! FLAG2 is set
               .
IF (BTEST (CLUSTER,FLAG3)) THEN
               .
               . ! FLAG3 is set
               .
IF (BTEST (CLUSTER,FLAG4)) THEN
               .
               . ! FLAG4 is set
               .
```

The intrinsic function BTEST, unlike the other intrinsic bit functions, is not restricted to words and longwords. In the following example, the privilege mask of the current process is examined to determine whether or not the process has the SETPRV privilege.

```
INCLUDE '($JPIDEF)'  ! Defines JPI$_ request codes
INCLUDE '($PRVDEF)'  ! Defines PRV$_ privilege codes
INCLUDE '($SSDEF)'   ! Defines SS$_ return codes
INTEGER*4 PRIV_MASK (2)
! Declare system procedure
INTEGER*4 LIB$GETJPI
STATUS = LIB$GETJPI (JPI$_CURPRIV,
2                    ,,
2                    PRIV_MASK,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Signal if SETPRV is not enabled
IF (.NOT. (BTEST (PRIV_MASK(1),PRV$V_SETPRV)) )
2    CALL LIB$SIGNAL (%VAL(SS$_NOPRIV))
```

1.6.2 Addresses

The following data types are valid in FORTRAN: byte, character, numeric, and logical data. However, in FORTRAN, a piece of data cannot be expressed as an address (other languages do understand this data type). Since system-defined procedures are not written exclusively for FORTRAN, a number of procedures return the address of data rather than the data itself. The steps for accessing data from an address differ depending on whether you have numeric or character data.

1.6.2.1 Numeric Data

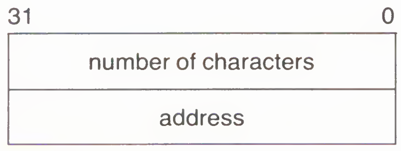
Given an address, use the following steps to access numeric data at that address.

1. Use the by value passing mechanism to pass the address to a subprogram.
2. In the subprogram, declare the dummy argument associated with the address as a numeric variable of the required size.
3. Reference the dummy argument to access the data.

1.6.2.2 Character Data

Given an address, use the following steps to access character data at that address.

1. Create a descriptor of the form



ZK-2029-84

2. Use the by reference passing mechanism (the default) to pass the descriptor to a subprogram.
3. In the subprogram, declare the dummy argument associated with the descriptor as a character variable of the required size or as a passed-length character string.
4. Reference the dummy argument to access the data.

In the following FORTRAN example, SORTING.FOR sorts a file and then uses the SOR$STAT routine to determine what version of the Sort Utility was used. SOR$STAT returns the address of a counted character string containing the version number, SORTING.FOR passes that address to GET_VERSION, and GET_VERSION can then access the counted string containing the version number. (The first byte of a counted character string contains the number of characters in the string.)

SORTING.FOR

```
CHARACTER*256  FILENAME_IN,
2              FILENAME_OUT
INTEGER*4      FN_SIZE_IN,
2              FN_SIZE_OUT
! To get version number
INTEGER*4    IDENT,
2            VERSION(2)
CHARACTER*10 V_STRING
EXTERNAL     SOR$K_IDENT
! SORT key information
INTEGER*2 KEY_BUFFER(5)
                       .
                       .
                       .
! Declare status and system routines
INTEGER STATUS,
2       SOR$INIT_SORT,
2       SOR$PASS_FILES,
2       SOR$BEGIN_SORT,
2       SOR$SORT_MERGE,
2       SOR$END_SORT,
2       SOR$STAT
                       .
                       .
                       .
! Pass files to SORT
STATUS = SOR$PASS_FILES (FILENAME_IN (1:FN_SIZE_IN),
2                        FILENAME_OUT (1:FN_SIZE_OUT))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Give SORT key information
STATUS = SOR$BEGIN_SORT (KEY_BUFFER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Start sorting
STATUS = SOR$SORT_MERGE ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

```
! Get SORT version
STATUS = SOR$STAT (%LOC(SOR$K_IDENT),
2                  IDENT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create a descriptor and pass it to GET_VERSION
VERSION(1) = 12
VERSION(2) = IDENT
CALL GET_VERSION (VERSION,
2                 V_STRING)
! End SORT
STATUS = SOR$END_SORT ()
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
                          .
                          .
                          .
```

GET_VERSION.FOR

```
SUBROUTINE GET_VERSION (VERSION,
2                       V_STRING)
! Dummy arguments
CHARACTER*(*) VERSION,
2             V_STRING
! VERSION length
INTEGER*4 LENGTH
! Intrinsic routine
INTRINSIC ICHAR
LENGTH = ICHAR (VERSION(1:1))
V_STRING = 'Sorted using version '
2          //VERSION (2:LENGTH)//' of SORT'
END
```

1.6.3 Buffers

In cases where many pieces of information are returned in a single buffer, define the buffer variable as a record; each field within the record contains one piece of information.

The system service SYS$CREPRC creates a process and, optionally, writes a message to a mailbox when the created process is deleted. The termination message is a buffer containing various pieces of information about the deleted process. To read the information from the message, make the message variable a record with separate fields for each piece of information. Since you must read the message from the mailbox using formatted I/O, it is easiest to create a record that is alternately defined as a character string or a sequence of fields. You can read the message into the character string field, and then use the other field definitions to reference the information.

```
STRUCTURE /TERM_MESSAGE/
 UNION
  MAP
   INTEGER*2    MSGTYP,
2               MSGSIZ
   INTEGER*4    FINALSTS,
2               PID,
2               JOBID,
2               TERMTIME(2)
   CHARACTER*8  ACCOUNT
   CHARACTER*12 USERNAME
   INTEGER*4    CPUTIM,
2               PAGEFLTS,
2               PFGLPEAK,
2               WSPEAK,
2               BIOCNT,
2               DIOCNT,
2               VOLUMES,
2               LOGIN(2),
2               OWNER
  END MAP
  MAP
   CHARACTER*84 MESSAGE
  END MAP
 END UNION
END STRUCTURE
```

The following FORTRAN program creates a process, reads the termination message when the created process is deleted, and displays a line of information about the deleted process.

```
! Status variable
INTEGER STATUS,
2       SYS$CREMBX,
2       SYS$GETDVIW,
2       SYS$CREPRC,
2       SYS$ASCTIM
! Define termination buffer
STRUCTURE /TERM_MESSAGE/
 UNION
  MAP
   INTEGER*2    MSGTYP,
2               MSGSIZ
   INTEGER*4    FINALSTS,
2               PID,
2               JOBID,
2               TERMTIME(2)
   CHARACTER*8  ACCOUNT
   CHARACTER*12 USERNAME
   INTEGER*4    CPUTIM,
2               PAGEFLTS,
2               PFGLPEAK,
2               WSPEAK,
2               BIOCNT,
2               DIOCNT,
2               VOLUMES,
2               LOGIN(2),
2               OWNER
  END MAP
  MAP
   CHARACTER*84 MESSAGE
  END MAP
 END UNION
END STRUCTURE
RECORD /TERM_MESSAGE/ TERMINATION
! String for termination time
CHARACTER*24 TERM_STRING
INTEGER*4    TERM_LEN
```

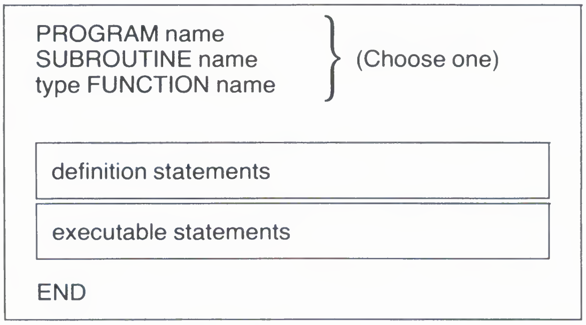
```
! Declare itmlst for $GETDVI
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
RECORD /ITMLST/ DVI_LIST(2)
! Buffers for $GETDVI
INTEGER*4 UNIT,
2         UNIT_LEN
! Item code for $DVIDEF
EXTERNAL DVI$_UNIT
! Channel for termination mailbox
INTEGER*2 MBX_CHAN
! Create mailbox
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,
2                    ,,,,
2                    'MBX_NAME')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get mailbox unit number
! Set up for $GETDVI
DVI_LIST(1).BUFLEN = 4
DVI_LIST(1).CODE = %LOC(DVI$_UNIT)
DVI_LIST(1).BUFADR = %LOC(UNIT)
DVI_LIST(1).RETLENADR = %LOC(UNIT_LEN)
DVI_LIST(2).END_LIST = 0
! Call $GETDVI
STATUS = SYS$GETDVIW (,
2                     %VAL(MBX_CHAN),
2                     ,
2                     DVI_LIST,
2                     ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Create process, passing mailbox unit number
STATUS = SYS$CREPRC (,
2                    'SYS$USER:[ACCOUNT]ADDEND',
2                    ,,,,,,
2                    %VAL(4),,
2                    %VAL(UNIT),)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Read termination mailbox
OPEN (UNIT = 2,
2     FILE = 'MBX_NAME',
2     CARRIAGECONTROL = 'LIST',
2     STATUS = 'NEW')
READ (UNIT=2,
2     FMT = '(A)') TERMINATION.MESSAGE
```

```
! Convert termination time to ASCII
STATUS = SYS$ASCTIM (TERM_LEN,
2                    TERM_STRING,
2                    TERMINATION.TERMTIME,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Display informational message
TYPE *,'Process ',TERMINATION.PID,' completed at ',
2      TERM_STRING(1:TERM_LEN),'.'
TYPE *,'Final status was ',TERMINATION.FINALSTS,'.'
END
```

2 Intraunit Logic and Local Storage

A program unit consists of either the main program or a subprogram. Storing each program unit, or set of related program units, in separate files provides a clear method of organization for complex programs.

The figure below outlines the general structure of a program unit.



ZK-2030-84

The first statement of a FORTRAN program unit is either a PROGRAM (optional, but recommended), SUBROUTINE, or FUNCTION statement that identifies the program unit by name and (for functions) data type. Program unit names must be unique among other program unit and common block names used within the program.

Following the first statement are definition statements, which mainly identify data elements and determine how they can be used. The next group of statements, the executable statements, performs the actual work of the program unit. The first executable statement in a program unit is called the program unit's entry point. The last statement of a FORTRAN program unit must be the END statement.

The following function, named CALC_SUMS, performs various calculations when invoked from another program unit.

CALC_SUMS.FOR

```
! ******************
! Function statement
! ******************
INTEGER FUNCTION CALC_SUMS (TOTAL_HOUSES,
2                            PERSONS_HOUSE,
2                            ADULTS_HOUSE,
2                            INCOME_HOUSE,
2                            AVG_PERSONS_HOUSE,
2                            AVG_ADULTS_HOUSE,
2                            AVG_INCOME_HOUSE,
2                            AVG_INCOME_PERSON,
2                            MED_INCOME_HOUSE)
! A function to calculate averages and median
! *********************
! Definition statements
! *********************
! Declare dummy arguments
INTEGER TOTAL_HOUSES,
2       MED_INCOME_HOUSE
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048),
2    AVG_PERSONS_HOUSE,
2    AVG_ADULTS_HOUSE,
2    AVG_INCOME_HOUSE,
2    AVG_INCOME_PERSON
! Declare control variables
REAL PERSONS,
2    ADULTS,
2    INCOME
INTEGER MEDIAN (101)
! Declare status variables and values
INTEGER STATUS_OK
PARAMETER (STATUS_OK = 1)
! *********************
! Executable statements
! *********************
! Calculate totals
DO I = 1, TOTAL_HOUSES
  PERSONS = PERSONS + PERSONS_HOUSE (I)
  ADULTS = ADULTS + ADULTS_HOUSE (I)
  INCOME = INCOME + INCOME_HOUSE (I)
! Divide and truncate to integer
  J = INCOME_HOUSE (I) / 1000
! Incomes under $1000 equal $1000
  IF (J .EQ. 0) J = 1
! Incomes over $100,000 not distinguished
  IF (J .GT. 101) J = 101
! Count incomes in that group
  MEDIAN (J) = MEDIAN (J) + 1
END DO
```

```
! Calculate averages
AVG_PERSONS_HOUSE = PERSONS / TOTAL_HOUSES
AVG_ADULTS_HOUSE = ADULTS / TOTAL_HOUSES
AVG_INCOME_HOUSE = INCOME / TOTAL_HOUSES
AVG_INCOME_PERSON = INCOME / PERSONS
! Calculate median income per house
J = 1
DO I = 2, 101
  IF (MEDIAN (I) .GT. MEDIAN (J)) J = I
END DO
MED_INCOME_HOUSE = J * 1000
! Return
CALC_SUMS = STATUS_OK
! *************
! End statement
! *************
END ! of subroutine
```

2.1 Local Storage

Local storage is the space allocated for data that is available only within the defining program unit. It does not include space allocated for data defined by, or available to, other program units. For example, in a FORTRAN program, data in an INTEGER statement uses local storage; data in a COMMON statement does not.

2.1.1 Data Type Definition

FORTRAN allows you to define the data type of an element in one of two ways (see the data type definitions which apply to your language):

- Explicitly—Define the data type of a variable by declaring the variable in a type declaration statement. (Chapter 6 describes each data type and its type declaration statement.)
- Implicitly—Define the data type of a variable implicitly by using the variable in an execution statement without first declaring it in a type declaration statement. By default, FORTRAN implicitly defines variables beginning with the letters I, J, K, L, M, or N as INTEGER variables and all other variables as REAL*4.

The FORTRAN program segment below explicitly defines TOTAL_HOUSES as an INTEGER*4 variable, explicitly defines all variables in the REAL*4 declaration statement as REAL*4 variables, and implicitly defines I as an INTEGER variable.

```
! Declare variables to hold statistics
INTEGER*4 TOTAL_HOUSES       ! Number of  households
REAL*4 PERSONS_HOUSE (2048), ! Number of persons per household
2      ADULTS_HOUSE (2048),  ! Number of adults per household
2      INCOME_HOUSE (2048),  ! Gross income per household
2      PERSONS,              ! Total people
2      ADULTS,               ! Total adults
2      INCOME                ! Total gross income
! Calculate totals
DO I = 1, TOTAL_HOUSES
  PERSONS = PERSONS + PERSONS_HOUSE (I)
  ADULTS = ADULTS_HOUSE (I)
  INCOME = INCOME_HOUSE (I)
END DO
```

In FORTRAN, you can override the default implicit data types by using the IMPLICIT statement. The program segment below implicitly defines all variables beginning with the letters A, P, and I as REAL*4 variables and explicitly declares I as an INTEGER*4 variable. Explicit data types always override implicit data types. (This example is functionally equivalent to the previous example.)

```
! Implicit typing
IMPLICIT REAL*4 (A,P,I)
! DO control variable
INTEGER*4 I
! Calculate totals
DO I = 1, TOTAL_HOUSES
  PERSONS = PERSONS + PERSONS_HOUSE (I)
  ADULTS = ADULTS_HOUSE (I)
  INCOME = INCOME_HOUSE (I)
END DO
```

To override the use of all implicit data types, use an IMPLICIT NONE statement. If you specify IMPLICIT NONE, you cannot specify other IMPLICIT statements and must explicitly declare all variables. In serious programming efforts, use IMPLICIT NONE.

IMPLICIT statements, or the IMPLICIT NONE statement, must precede data definition statements. The use of implicit data types does not affect the data type of FORTRAN intrinsic functions, but does affect the data type of any other external function.

2.1.2 Variable Initialization

You can initialize variables at compile time (before the program unit is invoked) or at run time (each time the program unit is invoked).

- Compile time—Initialize a variable at compile time if you want the variable initialized exactly once. Variables initialized at compile time require space in the image file; therefore, consider initializing large arrays at run time instead.
- Run time—Initialize a variable at run time if you want the variable reinitialized each time the program unit executes.

Unless you explicitly state otherwise, all variables used in FORTRAN programs are initialized to zero. However, in serious programming efforts, you should not depend on these zero values since they may change in future versions of FORTRAN or the VAX/VMS operating system.

2.1.2.1 One Time Initialization

In a FORTRAN program, to initialize a variable exactly once, use the DATA statement. The DATA statement assigns a value to a variable at compile time. The first time the program unit executes the variable contains the value specified by the DATA statement. During subsequent executions of the same program unit, the variable reflects any changes made to the initial value. In the FORTRAN example below, DATA statements initialize an array (ACCTS), an integer variable (PAGE_COUNT), and a character variable (PAGE_HEADER).

```
! Declare variables
INTEGER PAGE_COUNT,
2       ACCTS (10)
CHARACTER*55 PAGE_HEADER
! Initialize variables
DATA PAGE_COUNT /1/
DATA ACCTS /10*5/
DATA PAGE_HEADER /'House No.
2    Persons-house   Adults-house   Income-house'/
```

The first DATA statement assigns a value of 1 to the variable PAGE_COUNT; the second, a value of 5 to each of the ten elements in the array ACCTS; and the third, a 55-character string to the variable PAGE_HEADER.

You can initialize part of an array by using an implied DO loop within the DATA statement. The example below uses an implied DO loop to initialize the first four elements of the array ACCTS to the value 2 (the fifth element is not initialized).

```
INTEGER ACCTS (5)
DATA (ACCTS(I), I = 1,4) /4*2/
```

You can also initialize variables in type declaration statements by specifying the initial value between slashes following the variable. Only the variable immediately preceding the specified value is initialized. To initialize an array in this way, include a value for every element in the array. (Values specified in type declaration statements, like those specified in DATA statements, are assigned to the variable at compile time.) The example below initializes the integer ELEMENTS to the value 2 and the two elements of the NAMES array to JACK and JOE (NEW_NAME is not initialized).

```
INTEGER*4 ELEMENTS /2/
CHARACTER*10 NEW_NAME,
2            NAMES(2) /'JACK','JOE'/
```

2.1.2.2 Reinitialization

When a FORTRAN program unit is invoked more than once during program execution, its variables (with the exception of dummy arguments) are not reinitialized; that is, the variables retain the values they had at the end of the previous invocation of the program unit. To initialize a variable each time you invoke the program unit containing that variable, use an assignment or input statement in the executable portion of the program unit. The first executable statement shown in the example below assigns the value of 1 to the variable J. Each time the program unit is invoked, J is initialized to 1.

```
INTEGER*4 MEDIAN_INCOME_HOUSE,
2         MEDIAN (101)
                .
                .
                .
! Calculate median income per house
J = 1
DO I = 2, 101
  IF (MEDIAN (I) .GT. MEDIAN (J)) J = I
END DO
MED_INCOME_HOUSE = J * 1000
```

2.1.3 Named Constants

You can assign symbolic names to constant values by using a PARAMETER statement in the definition section of a program unit. Within the program unit containing the PARAMETER statement, you can use the symbolic name wherever you would use the constant. The PARAMETER statement is particularly useful for naming values that would otherwise be meaningless to someone reading the program. For example, the following PARAMETER statement assigns the symbolic name STATUS_OK to the integer value 1.

```
INTEGER STATUS_OK
PARAMETER (STATUS_OK = 1)
```

In the example below, the PARAMETER statement assigns the symbolic name PAGE_HEADER to the specified character string.

```
CHARACTER*(*) PAGE_HEADER
PARAMETER (PAGE_HEADER ='House No.
2     Persons-house   Adults-house   Income-house')
```

Note that you can use the passed-length notation, CHARACTER*(*), to define the length of a character string used in a PARAMETER statement. In the previous example, the CHARACTER variable PAGE_HEADER assumes the length of the character constant specified in the PARAMETER statement.

2.1.4 Equivalent Variables

Equivalent variables permit two or more variables to refer to the same area in memory. Use the FORTRAN EQUIVALENCE statement to equivalence variables that are in the same program unit.

The EQUIVALENCE statement causes each of the variables in a parenthesized list to be allocated storage beginning at the same location in memory. For example, the following EQUIVALENCE statement associates the variables UIC and UIC_LOAD, making the INTEGER*2 array UIC_LOAD refer to the same location in memory as the INTEGER*4 variable UIC.

```
INTEGER*2 UIC_LOAD (2)
INTEGER*4 UIC
EQUIVALENCE (UIC,UIC_LOAD)
```

You can make variables of different data types equivalent, and you can make an element of one array equivalent to an element of another array. When you equivalence elements of two arrays, the other elements of the two arrays may also be made equivalent. You can equivalence a variable not in a common block to a variable that is in a common block, but you cannot equivalence two variables in the same common block. You cannot equivalence a dummy argument to any other argument.

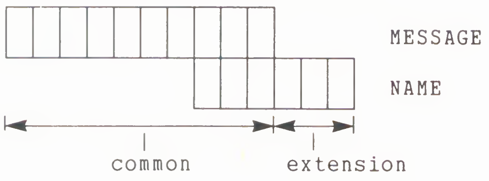
If you equivalence two arrays, the shorter of which is in a common block, the common block is automatically extended to include all of the elements in the longer array. However, since the starting address of a common block cannot change, arrays cannot be equivalenced if the equivalence would force an extension of the common block to precede the existing starting address.

Valid Equivalence:

```
BYTE MESSAGE (10)
CHARACTER NAME (6)
COMMON /MAIL_BOX/ MESSAGE
EQUIVALENCE (NAME,MESSAGE(8))
```

Valid equivalence:

```
BYTE MESSAGE (10)
CHARACTER NAME (6)
COMMON /MAIL_BOX/ MESSAGE
EQUIVALENCE (NAME,MESSAGE(8))
```



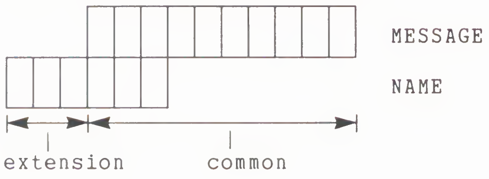
ZK-2031-84

Invalid Equivalence:

```
BYTE MESSAGE (10)
CHARACTER NAME (6)
COMMON /MAIL_BOX/ MESSAGE
EQUIVALENCE (NAME(4),MESSAGE)
```

Invalid equivalence:

```
BYTE MESSAGE (10)
CHARACTER NAME (6)
COMMON /MAIL_BOX/ MESSAGE
EQUIVALENCE (NAME(4),MESSAGE)
```



ZK-2032-84

2.1.5 Contiguous Storage

Variables are not necessarily stored in virtual memory in the order of their declaration. If the order of storage is significant (for example, if you are creating a data structure), define the variables in a common block or an array in the order of storage desired.

2.1.6 Large Data Structures and Dynamic Storage

All data storage is allocated statically; that is, memory is allocated when the data is given a value and remains allocated for the duration of the program's execution. Static storage does not require space in the image file (the file containing the executable program) unless a data element is initialized (for example, in a DATA statement). However, all data storage does require virtual memory.

Note

The size of virtual memory is determined by the system parameter VIRTUALPAGECNT . Half of virtual memory is reserved for system use and is usually inaccessible to you. Of the remaining half, half is automatically allocated to you and half you can dynamically allocate.

Generally, you have enough virtual memory to ensure sufficient space for your variables. For example, if you estimate a need for storing up to 2048 real numbers each in the variables PERSONS_HOUSE, ADULTS_HOUSE, and INCOME_HOUSE, simply allocate arrays with dimension 2048.

```
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048)
```

If you are approaching your virtual memory limit, consider making the data structures smaller and processing your data in pieces rather than all at once. If you are using large arrays or character strings that you do not need for the duration of the program, dynamically allocate and deallocate the storage for those variables by using the run-time library procedures LIB$GET_VM and LIB$FREE_VM.

Dynamic allocation and deallocation of storage is especially recommended for general purpose utility subprograms, since the subprogram cannot anticipate the storage needs of the main program and should not burden the image with unnecessary static storage. The procedure LIB$GET_VM allocates a requested amount of storage (in bytes) and returns the address of the first byte so allocated. The procedure LIB$FREE_VM deallocates storage allocated by LIB$GET_VM. You cannot use the address returned by LIB$GET_VM directly in FORTRAN, but you can pass it to a subprogram so that the subprogram can access the allocated storage.

2.1.6.1 Dynamic Storage of Numeric Data

To allocate and deallocate storage for numeric data dynamically do the following:

1. Put the code that uses the storage in a subprogram.
2. Call LIB$GET_VM to obtain the address of a specified number of bytes of storage.
3. Invoke the subprogram that uses the storage and pass it the address returned by LIB$GET_VM and the number of allocated bytes. You must pass the address using the built-in function %VAL.
4. In the subprogram, define the dummy argument that is associated with the address of the storage as a numeric variable equal in size to the number of allocated bytes. Typically, you declare the dummy argument as an assumed-size or adjustable array of the required data type.
5. In the invoking program unit, deallocate the storage by calling LIB$FREE_VM.

In the example below, the invoking program unit invokes LIB$GET_VM to get 512 bytes of storage and then passes the address and size of the storage area to the subprogram that uses the storage.

Invoking Program Unit

```
! Program to get dynamic storage
PROGRAM GETMEM

! Declare variables
INTEGER STATUS,
2       VM_SIZE,  ! Size of storage in bytes
2       VM_ADDR   ! Address of storage

! Declare user routine
INTEGER USEMEM
! Declare library routines
INTEGER LIB$GET_VM,
2       LIB$FREE_VM
! Get storage
VM_SIZE = 512
STATUS = LIB$GET_VM (VM_SIZE,
2                    VM_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Use storage
STATUS = USEMEM (%VAL(VM_ADDR),
2                VM_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

```
! Free storage
STATUS = LIB$FREE_VM (VM_SIZE,
2                     VM_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
                         .
                         .
                         .
```

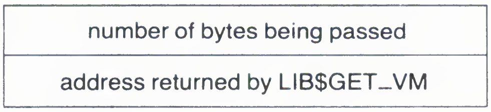
USEMEM.FOR

```
INTEGER FUNCTION USEMEM (ARRAY,
2                        ARRAY_SIZE)
! Subprogram to use dynamic storage

! Declare dummy arguments
INTEGER*4 ARRAY_SIZE,
2         ARRAY(ARRAY_SIZE/4)

! Set good return status
USEMEM = 1
                    .
                    .
                    .
```

2.1.6.2 Dynamic Storage of Character Data

To allocate and deallocate storage for character data dynamically:

1. Put the code that uses the storage in a subprogram.
2. Call LIB$GET_VM to obtain the address of a specified number of bytes of storage.
3. Invoke the subprogram that uses the data and pass it a descriptor of the character data. The descriptor is a quadword in the format



ZK-2033-84

4. In the subprogram, make the dummy argument that is associated with the descriptor a passed-length character string or CHARACTER variable of appropriate length. If you specify the dummy argument as a CHARACTER array, use an assumed-size or adjustable array.
5. In the invoking program unit, deallocate the storage by calling LIB$FREE_VM.

In the FORTRAN example below, the invoking program unit passes the descriptor as the array VM_DESC: the first element contains the size (VM_SIZE) and the second contains the address (VM_ADDR).

Invoking Program Unit

```
! Program to get dynamic storage
PROGRAM GETMEM
! Declare variables
INTEGER STATUS,
2       VM_SIZE,  ! Size of storage in bytes
2       VM_ADDR   ! Address of storage
! Declare user routine
INTEGER USEMEM
! Declare library routines
INTEGER LIB$GET_VM,
2       LIB$FREE_VM
! Declare descriptor
INTEGER VM_DESC (2)
! Get storage
VM_SIZE = 512
STATUS = LIB$GET_VM (VM_SIZE,
2                    VM_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Use storage
VM_DESC(1) = VM_SIZE
VM_DESC(2) = VM_ADDR
STATUS = USEMEM (VM_DESC)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Free storage
STATUS = LIB$FREE_VM (VM_SIZE,
                      VM_ADDR)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
                        .
                        .
                        .
```

USEMEM.FOR

```
INTEGER FUNCTION USEMEM (ARRAY)
! Subprogram to use dynamic storage
! Declare dummy arguments
CHARACTER*8 ARRAY(*)
! Set good return status
USEMEM = 1
                    .
                    .
                    .
```

2.1.6.3 Dynamic Storage Statistics

You can check the number of calls made to both LIB$GET_VM and to LIB$FREE_VM and the number of bytes allocated and deallocated by using either LIB$STAT_VM or LIB$SHOW_VM. To obtain one statistic in binary form, use LIB$STAT_VM. To obtain one or all of the statistics formatted as a character string, use LIB$SHOW_VM. The following example calls LIB$SHOW_VM.

```
! Show storage used
STATUS = LIB$SHOW_VM()
```

LIB$SHOW_VM returns a message similar to the following. (By default, LIB$SHOW_VM writes the message to SYS$OUTPUT; however, you can write a subprogram that changes that default. For details, see the description of LIB$SHOW_VM in the *VAX/VMS Run-Time Library Routines Reference Manual*).

```
3 LIB$GET_VM calls,0 LIB$FREE_VM calls,952 bytes still allocated
```

Note

FORTRAN I/O routines obtain storage dynamically. The statistics returned by LIB$SHOW_VM include FORTRAN's calls to LIB$GET_VM and LIB$FREE_VM.

2.2 Serial Execution

Statements are normally executed in the order in which they occur in the program unit—serially, from the entry point of the program unit to its END statement. In order to maintain clarity of structure in a program unit do the following:

- Make each operation a visibly separate block of code (separated with comments or blank lines) and prefix each block with explanatory comments.
- Factor out complex pieces of code by making them subprograms.

For example, the following executable statements from the function CONVERT.FOR are organized by operation.

```
! Initialize status
STATUS = STATUS_OK
! Convert persons per house
READ (UNIT = FIX_PERSONS_STRING,
2     FMT = INTEGER_FMT) FIX_PERSONS_HOUSE
! Convert adults per house
READ (UNIT = FIX_ADULTS_STRING,
2     FMT = INTEGER_FMT) FIX_ADULTS_HOUSE
! Convert income per house
READ (UNIT = FIX_INCOME_STRING,
2     FMT = INTEGER_FMT) FIX_INCOME_HOUSE
CONVERT_FIXES = STATUS
END ! Of function
```

In FORTRAN, you can circumvent serial execution with the ENTRY, RETURN **label**, and GOTO statements; although, such transfers of control are difficult to follow and are not recommended.

2.3 Conditional Execution

Conditional execution permits you to execute one or more statements (a statement block) depending upon a specified condition. If the condition is logically true, the statement block following it is executed; otherwise, control passes to the executable statement following the statement block.

In FORTRAN, conditional logic is implemented primarily through the statements IF, ELSE IF, and ELSE. (The FORTRAN DO WHILE statement also executes conditionally; see Section 2.4.). Other languages use different conditional expressions. See the appropriate programming manual for information on other conditional expressions.

2.3.1 Specifying the Condition

You can base execution of a statement block on one condition or multiple conditions. Each condition is an expression that evaluates to a logical value (Section 6.5 discusses logical data). In the FORTRAN example below, the READ statement is executed if the logical value STATUS is true.

```
IF (STATUS) THEN
  READ (UNIT = FIX_PERSONS_STRING,
2       FMT = INTEGER_FMT) FIX_PERSONS_HOUSE
END IF
```

The following statement assigns a value to STATUS if the logical expression is true.

```
IF (HOUSE_NO .GT. MAXSTATS) THEN
  STATUS = %LOC (INCOME_MAXSTATS)
END IF
```

In FORTRAN, to base execution of a statement on more than one condition, use the logical operators .AND., .OR., .XOR., .NEQV., and .EQV., as described in Section 6.5, to combine the conditions (see the logical operators available in your language). In the example below, execution of the statement block requires both that STATUS be logically true and that the value of LINE_COUNT be greater than 58.

```
IF ((STATUS) .AND. (LINE_COUNT .GT. 58)) THEN
     .
     . ! Statement block
     .
END IF
```

You can use any number of logical operators in an IF statement. In the example below, two conditions must be met for the statement block to execute: STATUS must be true, and either NEW_PAGE must be true or LINE_COUNT must be greater than 58.

```
! Write new STATS.SAV
IF ( (STATUS) .AND.
2    ((NEW_PAGE) .OR. (LINE_COUNT .GT. 58)) ) THEN
    .
    . ! Statement block
    .
END IF
```

Parentheses force the order of interpretation. Even if the order does not need to be forced, parentheses make clear the order in which conditions are to be interpreted—from the innermost parentheses to the outermost.

2.3.2 Single Conditional Block

The simplest form of an IF statement, a logical IF statement, tests a condition and executes one statement if the condition is true.

```
IF (condition) statement
```

The statement following the condition can be any executable statement except a DO, END DO, END, or IF statement. In the following example of a logical IF statement, LIB$ERASE_LINE executes only if the logical value STATUS is true.

```
IF (STATUS) STATUS = LIB$ERASE_LINE (LINE_NO,
2                                    END_COLUMN)
```

A block IF statement tests a condition and executes one or more statements if the condition is true.

```
IF (condition) THEN
    .
    . ! Statement block
    .
END IF
```

In a block IF statement, the keyword THEN must follow the condition and the END IF statement must terminate the block of statements, as shown in the example below.

```
! Load fix values into arrays
IF (STATUS) THEN
  PERSONS_HOUSE (FIX_HOUSE_NO) = FIX_PERSONS_HOUSE
  ADULTS_HOUSE (FIX_HOUSE_NO) = FIX_ADULTS_HOUSE
  INCOME_HOUSE (FIX_HOUSE_NO) = FIX_INCOME_HOUSE
END IF
```

Indenting the statements in the IF statement block formats the IF block clearly, which is especially helpful with nested IF structures. Generally, since block IF statements are easier to read, they are preferred to logical IF statements.

2.3.3 Multiple Conditional Blocks

You can conditionally execute multiple statement blocks exclusively (executing only one statement block) or inclusively (executing more than one block).

2.3.3.1 Exclusive Conditional

An exclusive conditional construction permits you to execute one of several statement blocks. In its simplest form, the exclusive conditional construction allows you to execute one of two statement blocks. Following is the structure of statement blocks used in FORTRAN.

```
IF (condition) THEN
       .
       . ! Statement block
       .
ELSE
       .
       . ! Statement block
       .
END IF
```

If the condition is true, the first statement block is executed; otherwise, the second statement block is executed.

Another form of the exclusive conditional construction permits you to execute the first statement block that follows a true condition. If no condition is true, none of the statement blocks are executed. If more than one condition is true, only the first true statement block is executed.

```
IF (condition) THEN
       .
       . ! Statement block
       .
ELSE IF (condition) THEN
       .
       . ! Statement block
       . ! Optional ELSE IF statements
       .
END IF
```

In the example below, if the first condition is true, regardless of the second condition, STATUS is assigned the value of INCOME_CTRLZ. If the first condition is false and the second condition is true, STATUS is set to INCOME_FORIOERR. If neither condition is true, STATUS remains unchanged.

```
INTEGER*4 IOSTAT,
2          IO_OK,
2          EOF,
2          STATUS
EXTERNAL INCOME_FORIOERR,
2        INCOME_CTRLZ
        .
        .
        .
IF (IOSTAT .EQ. EOF) THEN
  STATUS = %LOC (INCOME_CTRLZ)
ELSE IF (IOSTAT .NE. IO_OK) THEN
  STATUS = %LOC (INCOME_FORIOERR)
END IF
```

In the final form of the exclusive conditional construction, the first statement block that follows a true condition is executed. If no condition is true, the statement block that follows the ELSE statement is executed.

```
IF (condition) THEN
       .
       . ! Statement block
       .
ELSE IF (condition) THEN
       .
       . ! Statement block
       . ! Optional ELSE IF statements
       .
ELSE
       .
       . ! Statement block
       .
END IF
```

The example below uses an exclusive conditional construction to call an appropriate subprogram depending on which command qualifier is entered on the command line.

```
! Get qualifier and dispatch to appropriate routine
! If user types /ENTER
IF (CLI$PRESENT ('ENTER')) THEN
  STATUS = GET_STATS (TOTAL_HOUSES,
            .
            . ! Statement block
            .
! If user types /FIX
ELSE IF (CLI$PRESENT ('FIX')) THEN
  STATUS = CLI$GET_VALUE ('FIX',
            .
            . ! Statement block
            .
! If user types /REPORT
ELSE IF (CLI$PRESENT ('REPORT')) THEN
  STATUS = REPORT (TOTAL_HOUSE,
            .
            . ! Statement block
            .
ELSE
  STATUS = %LOC (INCOME_NOACTION)
END IF
```

The statement that follows the ELSE statement is executed only if none of the qualifiers (/ENTER, /FIX, and /REPORT) are entered. Note that if /REPORT and /ENTER are both entered on the command line, only the first block (ENTER) is executed. (To execute both blocks, use the inclusive conditional construction described in Section 2.3.3.2.)

2.3.3.2 Inclusive Conditional

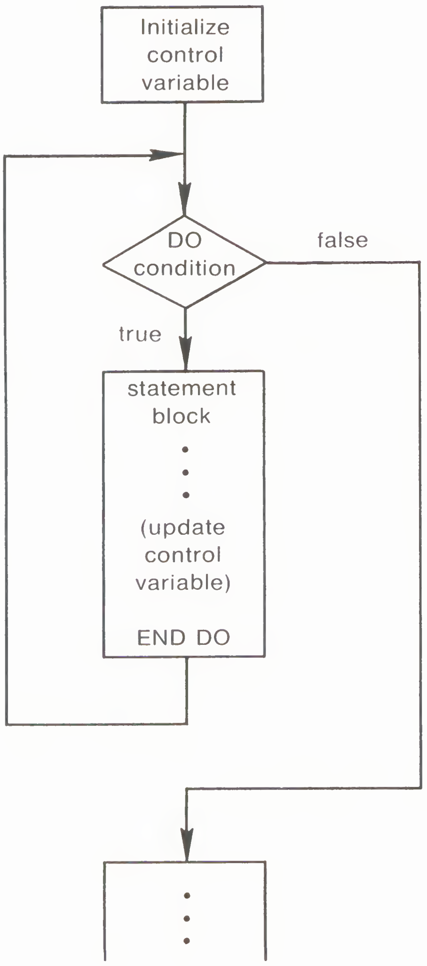
An inclusive condition contruction permits you to execute each statement block that follows a true condition. The example below calls the appropriate subprogram for each qualifier entered on the command line.

```
! Qualifier check
IF (CLI$PRESENT ('ENTER')) THEN
  STATUS = GET_STATS (TOTAL_HOUSES,
            .
            . ! Statement block
            .
END IF
IF (CLI$PRESENT ('FIX')) THEN
  STATUS = CLI$GET_VALUE ('FIX',
            .
            . ! Statement block
            .
END IF
IF (CLI$PRESENT ('REPORT')) THEN
  STATUS = REPORT (TOTAL_HOUSE,
            .
            . ! Statement block
            .
END IF
! Error check
IF ((.NOT. CLI$PRESENT ('ENTER')) .AND.
2   (.NOT. CLI$PRESENT ('FIX')) .AND.
2   (.NOT. CLI$PRESENT ('REPORT'))) THEN
  STATUS = %LOC (INCOME_NOACTION)
END IF
```

Since the inclusive conditional construction does not provide an ELSE statement whose statement block is executed in the event that no condition is true, you must check the negation of each condition specified in the inclusive construction, as shown in the final IF statement of the previous example. (Alternatively, you could set STATUS equal to INCOME_NOACTION at the beginning of the program segment; if none of the qualifiers are entered, STATUS remains equal to INCOME_NOACTION, otherwise it changes.)

2.4 Iterative Logic

Iterative logic repeatedly executes a statement block as long as a specified condition is true. Several forms of the DO statement implement iterative logic through the following basic loop structure.



ZK-1936-84

A test in the beginning of the statement block, or loop, determines whether or not the statements within the loop are executed. The variable that controls the outcome of that test, the control variable, is initialized before the loop and updated within the loop to eventually terminate execution of the loop.

2.4.1 DO WHILE Statement

The FORTRAN DO WHILE statement repeatedly executes the statements in the body of the loop while a specified condition is true. When the condition becomes false, control transfers to the next executable statement following the loop. The DO WHILE statement has the following general form:

```
(initialize control variable)
DO WHILE (condition)
      .
      . ! Statement block
      . ! Update control variable
END DO
```

The terminating statement of a DO loop must be END DO.

The example below continues to get a character from STRING while the character is blank and the value of FIRST_DIGIT is less than or equal to that of SIZE.

```
! Find first digit in number
FIRST_DIGIT = 1
DO WHILE ((STRING (FIRST_DIGIT:FIRST_DIGIT) .EQ. ' ') .AND.
2          (FIRST_DIGIT .LE. SIZE))
  FIRST_DIGIT = FIRST_DIGIT + 1
END DO
```

Note that FIRST_DIGIT is initialized to 1 before the loop and then incremented each time through the loop. If digit is not found, FIRST_DIGIT is equal to SIZE plus 1.

Indenting the statements in the DO statement block formats the DO block clearly. This is especially helpful with nested DO structures.

2.4.2 Indexed DO Statement

The indexed DO statement executes a statement or statement block a specified number of times. It has the following general form:

```
DO control = initial,terminal[,increment]
          .
          . ! Statement block
          .
END DO
```

The DO construction automatically initializes and updates the control variable. **Control**, which must be an integer or real variable, represents the control variable. **Initial** represents the value of **control** before the loop starts. **Increment**, which can be positive or negative, represents a value to be added to **control** after each iteration of the loop; **increment** defaults to +1. **Terminal** represents the value of **control** after which the loop terminates. **Initial**, **terminal**, and **increment** are numeric values.

The example below executes the statement block while I is less than or equal to the variable TOTAL_HOUSES. The control variable, I, starts with a value of 1 and increments by 1 each time through the loop. (Note that I is also used as a subscript for the arrays PERSONS_HOUSE and ADULTS_HOUSE.)

```
DO I = 1, TOTAL_HOUSES
  PERSONS = PERSONS + PERSONS_HOUSE (I)
  ADULTS = ADULTS + ADULTS_HOUSE (I)
END DO
```

The following DO WHILE statement is equivalent to the previous indexed DO statement.

```
I = 1
DO WHILE (I .LE. TOTAL_HOUSES)
  PERSONS = PERSONS + PERSONS_HOUSE (I)
  ADULTS = ADULTS + ADULTS_HOUSE (I)
  I = I + 1
END DO
```

2.5 Nesting

In FORTRAN, both IF constructions and DO loops can be nested within themselves and within each other. In either case, any nested construction must be entirely contained within the outer construction. In order to maintain clear code, indent each nested construction and comment each END statement.

Properly Nested:

```
IF (condition) THEN
     .
     .
     .
  DO WHILE (condition)
     .
     .
     .
  END DO
END IF
```

Improperly Overlapping:

```
IF (condition) THEN
     .
     .
     .
  DO WHILE (condition)
     .
     .
     .
END IF
  END DO
```

In the example below, an exclusive conditional construction is nested within a single condition block, which is nested within a DO WHILE.

```
DO WHILE (WHOOPS)
  READ (UNIT=STRING (1:STRING_SIZE),
2       FMT=INTEGER_FMT,
2       IOSTAT=IOSTAT) STAT

  ! Erase whoops
  STATUS = LIB$ERASE_LINE (LINE_NO,
2                          END_COLUMN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  WHOOPS = .FALSE.

  ! If I/O fails
  IF (IOSTAT .NE. IO_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .EQ. FOR$_INPCONERR) THEN

      ! Repeat until user provides convertible input
      ! (or presses CTRL/Z)
      STATUS = LIB$ERASE_LINE (LINE_NO,
2                              COLUMN_NO)
      IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
      STATUS = LIB$PUT_SCREEN ('whoops!',
2                              LINE_NO,
2                              END_COLUMN)
      IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
      WHOOPS = .TRUE.
    ELSE
      ! If unexpected read error
      IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    END IF  ! STATUS .EQ. FOR$_INPCONERR

  END IF  ! IOSTAT .NE. IO_OK

END DO  ! WHILE (WHOOPS)
```

3 Program Synchronization and Communication

Synchronization and communication 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. Since a process executes exactly one image at a time, you cannot execute two or more programs concurrently in a single process.

3.1 Creating Processes

Creating processes from within your program permits you to:

- Duplicate command-level functionality—Create a process that executes DCL commands.
- Perform parallel processing—Create a process that executes one part of your application while the parent process continues executing a different part.
- Implement multiuser applications—Create a process for each application user. The parent process can coordinate the input from the created processes.
- Specify a time for program execution—Create a process that executes a program, and then hibernate that process. By waking the created process at a specified time or timed intervals, you can choose the execution time of the program.
- Isolate code—Create a process that executes privileged or sensitive code.

A created process can be either a subprocess or a detached process. To create a subprocess, use the LIB$SPAWN Run-Time Library routine, which duplicates the functions of the DCL command SPAWN, or the SYS$CREPRC system service. To create a detached process, you must use SYS$CREPRC (see Section 3.1.3).

Usually you create subprocesses. However, a subprocess is dependent on its parent and is deleted when the parent process exits. Therefore, if you want a created process to continue executing a program after the parent exits, you must use a detached process.

Note

The *VAX/VMS DCL Dictionary* discusses the SPAWN command and the characteristics of spawned subprocesses. The following discussion assumes that you are familiar with that information.

3.1.1 Creating Subprocesses

You can create a subprocess using LIB$SPAWN or SYS$CREPRC. As shown in the following table, LIB$SPAWN creates a more complete context for a subprocess. (The table assumes that defaults were used in the calls to LIB$SPAWN and SYS$CREPRC.) The *VAX/VMS Run-Time Library Routines Reference Manual* and the *VAX/VMS System Services Reference Manual* contain complete descriptions of LIB$SPAWN and SYS$CREPRC.

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

[1]The created subprocess can include DCL by executing the system image SYS$SYSTEM:LOGINOUT.EXE (example in Section 3.1.3.).

[2]Plus group and job logical name tables.

3.1.1.1 Invoking LIB$SPAWN

Typically, when you invoke LIB$SPAWN, you use only the first four arguments.

- Command line (argument 1)—Specify a command to be executed in the created subprocess. To execute multiple commands, invoke a command procedure ('@command-procedure').
- SYS$INPUT and SYS$OUTPUT (arguments 2 and 3)—Specify the subprocess's equivalence names for SYS$INPUT and SYS$OUTPUT. If you omit these arguments, the subprocess inherits the equivalence names of the parent. If you specify SYS$INPUT as a file (or nonterminal device), the subprocess is created as a noninteractive process. These

arguments correspond to the /INPUT and /OUTPUT qualifiers of the DCL command SPAWN.

The logical names SYS$INPUT and SYS$OUTPUT are fully defined by these two arguments. They are not affected by bit 2 of the fourth argument, which defines other logical names for the subprocess.

- Context and execution (argument 4)—Specify a mask that indicates whether the subprocess inherits symbols, logical names, and/or keypad definitions from the parent, and whether the subprocess executes at the same time as the parent process or while the parent process hibernates.

The following statement 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')
```

3.1.1.2 Subprocess Context

It may take a few seconds for LIB$SPAWN to create a subprocess. You can decrease this time by indicating that the created subprocess should inherit only part of the parent's process context. For example, by specifying **flags** (argument 4) as CLI$M_NOLOGNAM you prevent the subprocess from inheriting the parent's logical name definitions. (Logical names required by the subprocess can be placed in the job logical name table LNM$JOB).

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 4.2.4.)

```
! 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)
```

3.1.1.3 Subprocess Execution

A subprocess can execute either while the parent process hibernates (in line) or while the parent process continues to execute (concurrent). Unless you specify otherwise, a subprocess executes in line. To execute a subprocess concurrently, specify the **flags** argument (argument 4) as CLI$M_NOWAIT bit when you invoke LIB$SPAWN. If the parent process must wait for information provided by the subprocess, execute the subprocess in line. Otherwise, consider executing the subprocess concurrently.

You can perform terminal I/O from a concurrently executing subprocess; however, to do so requires that you include code that synchronizes access to the terminal, as shown in Section 3.1.2. To avoid synchronization problems when you execute a subprocess concurrently, equivalence the subprocess's SYS$INPUT and SYS$OUTPUT to a file or device other than the terminal. (Specify **input-file** and **output-file** when you invoke LIB$SPAWN rather than allowing them to default to the parent's equivalence names; by default, the terminal.)

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 4.2.4.)

```
! 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
```

3.1.1.4 Debugging a Program in a Subprocess

A program should be debugged before you invoke it within a subprocess. However, in cases where you must debug within the subprocess, 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's DBG> prompt 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 will have to press RETURN to display the DBG> prompt. (By pressing RETURN, 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.)

3.1.2 Synchronizing Terminal I/O with the Lock Manager

When you specify the user's terminal as the I/O device for a concurrently executing subprocess, use the lock manager to ensure that only one process accesses the terminal at any one time. Since the parent process and the created subprocess must use the lock manager system service to synchronize terminal I/O, the subprocess must execute a user-written program that includes the synchronization rather than a list of DCL commands.

The lock manager allows cooperating processes to synchronize access to a shared resource (for example, a file, program, or device). When cooperating processes reference a resource, they do so by an agreed-upon name. The lock manager works only with that name; it has no control over the resource itself. The lock manager can only grant or refuse a lock request; it cannot ensure that processes use the lock manager or respect the locks placed on a resource.

To request access to a resource, use the SYS$ENQ or SYS$ENQW system service 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.) Six lock modes allow a process to indicate the extent to which it is willing to share the resource. For example, a null lock allows any other process read and/or write access to the resource; an exclusive lock allows no other process access to the resource.

The *VAX/VMS System Services Reference Manual* describes the SYS$ENQW system service and its arguments. Typically, when you invoke SYS$ENQW you use only the following arguments:

- Lock mode (argument 2)—Specify the requested lock mode. When the shared resource is a terminal, only two lock modes are significant: null (LCK$K_NLMODE) and exclusive (LCK$K_EXMODE).
- Status block (argument 3)—Specify two words and a longword. The first word receives the final status of the lock request and the second word is a place holder. The first time you reference a resource, the longword receives an identification number for the resource. From then on, to reference the resource, you use the lock status block to pass the identification number of the resource to the lock manager.
- Options (argument 4)—Specify a mask with the LCK$V_CONVERT bit set to indicate that the lock manager should change the mode of the lock.
- Resource name (argument 5)—Specify the name of the resource.

The following program segment 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.)

```
! 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))
```

To share a terminal between a parent process and a subprocess, each process requests a null lock on a shared resource name. Then, each time one of the processes wants to perform terminal I/O, it requests an exclusive lock, performs the I/O, and requests a null lock.

Since the lock manager is only effective between cooperating programs, 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 that event flag has been 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.)

After the parent process exits, a created process cannot synchronize access to the terminal and should use the SYS$BRKTHRU system to write to the terminal.

3.1.3 Creating Detached Processes

In general, you should create a detached process only if you have a program that must continue executing after the parent process logs out.

The *VAX/VMS System Services Reference Manual* contains a complete description of the SYS$CREPRC system service and its arguments. Typically, when you invoke the SYS$CREPRC system service, you specify only a subset of the following arguments:

- Image (argument 2)—Specify the name of the image to be executed by the created process. This argument is required.

  The created process is not a DCL-based process; therefore, it does not understand DCL commands. To execute DCL commands from a process created by SYS$CREPRC, you must: 1) specify SYS$SYSTEM:LOGINOUT.EXE as the image to execute and 2) equate SYS$INPUT to a command procedure containing the commands that you want executed.

- SYS$INPUT and SYS$OUTPUT (arguments 3 and 4)—Specify the equivalence names for SYS$INPUT and SYS$OUTPUT. If you specify a device, you must specify the device name; no logical name translations are performed. If these arguments are omitted, SYS$INPUT and SYS$OUTPUT are undefined.
- Process name (argument 8)—Specify a name for the created process.
- Priority (argument 9)—Specify the base priority of the created process. Since priority defaults to 0, you should specify this argument. Unless you are working on a real-time application, use a priority of 4.
- Options (argument 12)—Allows you to indicate the characteristics of the created process (for example, to specify that the created process be 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
```

To write to another process's terminal from a detached process, use the SYS$BRKTHRU system service.

3.1.4 Specifying a Time for Program Execution

To execute a program at a specified time or at timed intervals, create a subprocess or detached process, hibernate the created process, and then wake the process at the required times. A subprocess is deleted when the parent process exits; therefore, if you expect the parent process to exit before the program in the created process finishes executing, create a detached process rather than a subprocess.

3.1.4.1 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 will not want the parent process to remain in hibernation until the process executes, execute the process concurrently.

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

The following example executes a program at a specified 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 6.11 discusses time manipulation).

```
! 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
```

3.1.4.2 Timed Intervals

To execute a program at timed intervals, you can use either LIB$SPAWN or LIB$CREPRC. With LIB$SPAWN, create a subprocess that executes a command procedure containing three commands: the DCL command WAIT, the command that invokes the desired program, and a GOTO command that directs control back to the WAIT command. Since you will not want the parent process to remain in hibernation until the subprocess executes, execute the subprocess concurrently. See the previous section for an example of LIB$SPAWN.

The following steps describe how to use SYS$CREPRC to execute a program at timed intervals. To create a detached process, you must use SYS$CREPRC.

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 the subprocess and a time interval at which the system should repeat the wakeup call.

The following example 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 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.

```
! 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))
                                 .
                                 .
                                 .
```

3.2 Examining and Modifying Processes

The following routines allow you to create, modify, and examine processes.

LIB$GETJPI	Returns process information
SYS$GETJPI	Returns process information
SYS$GETJPIW	Returns process information
SYS$SETPRN	Sets process name
SYS$SETPRI	Sets process priority
SYS$SETSWM	Controls swapping of process

By default, these routines reference the current process. To reference another process, you must specify either the process identification number or the process name when you invoke the routine. 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.

3.2.1 Examining Processes

Typically, you use the LIB$GETJPI Run-Time Library routine to return information about a process. The *VAX/VMS Run-Time Library Routines Reference Manual* contains a complete description of this routine including the various items of information that you can request. LIB$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.

Note that a few of the items listed in the LIB$GETJPI description can be returned as longwords and/or strings. (SYS$GETJPI and SYS$GETJPIW return these items only as longword values.) In the following example, the string argument rather than the numeric argument is specified so LIB$GETJPI returns 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$GETJPI. Call LIB$GETJPI repetitively until it returns a status of SS$_NOMOREPROC indicating that all processes on the system have been examined.

The following program creates a file, PROCNAME.RPT, that lists 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.

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

LIB$GETJPI provides an easy method of obtaining a single item of information. To request many items of information about a process, you can make multiple calls to LIB$GETJPI, or use SYS$GETJPI or SYS$GETJPIW. SYS$GETJPI executes asynchronously and SYS$GETJPIW executes synchronously; other than that the two routines are the same (see Section 3.3.1.3).

To specify a list of items for SYS$GETJPI or SYSGETJPIW (even if that list contains only one item), use a record structure as shown in Section 1.5.8. The following example uses SYS$GETJPIW to request the process name and user name associated with the process whose process identification number is in SUBPROCESS_PID.

```
! 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
! 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
                       .
                       .
                       .
```

3.2.2 Monitoring Program Execution

When you execute a lengthy image, monitoring its execution is difficult, especially if the image does not display information to the terminal. However, if you use the SYS$SETPRN system service to change the process name at significant points in your program, you can easily monitor the program's progress with either CTRL/T (if the image is currently executing in your process) or the DCL command SHOW SYSTEM (if the image is executing in a detached process, such as a batch job).

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))
                        .
                        .
                        .
```

3.2.3 Controlling Process Scheduling

In general, you should let the system control process scheduling. However, if necessary, you can inhibit swapping and/or increase the priority of a particular process.

- 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).

  Inhibit swapping for any process with the SYS$SETSWM system service. If you create a subprocess with the LIB$SPAWN routine, you can inhibit swapping by executing the DCL command SET PROCESS/NOSWAP as the first command in a command procedure. If you create a process with the SYS$CREPRC system service, you can inhibit swapping for the process by setting the inhibit bit in the **stsflg** argument (see the SYS$CREPRC description in the *VAX/VMS System Services Reference Manual*). A process must have PSWAPM privilege to inhibit swapping.

- 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.

  Set the base priority for any process with the SYS$SETPRI system service. If you create a subprocess with the LIB$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. If you create a process with the SYS$CREPRC system service, you can set the base priority of the process with the **baspri** argument. You must have ALTPRI privilege to increase a process's base priority.

3.3 Controlling Program Execution

Event flags and asynchronous system traps (ASTs) allow you to synchronize separate operations.

3.3.1 Synchronizing Operations with Event Flags

Event flags are divided into four clusters: two for local event flags and two for common event flags.

- Local event flags—Local event flags are process specific; use them to synchronize events within a program or to pass information from the current image to an image executed later by the same process (Section 3.5 discusses intraprocess communication). 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.

Cluster	Flags
0	0—31[1]
1	32—63

[1]Event flags 24—31 are reserved for the system.

- Common event flags—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). Before a program can reference a common event flag, it must associate a cluster name with a common event flag cluster (see Section 3.3.1.2).

Cluster	Flags
2	64—95
3	96—127

3.3.1.1 Manipulating Event Flags

The following system-defined procedures allow you to manipulate event flags. The last four system services listed place your process into a local event flag (LEF) or a common event flag (CEF) wait state depending on the event flag(s) specified.

LIB$GET_EF	Returns the number of a local event flag not currently in use
LIB$FREE_EF	Frees a local event flag number
SYS$CLREF	Clears an event flag
SYS$SETEF	Sets an event flag
SYS$READEF	Returns the value of one cluster
SYS$SYNCH	Waits for an event flag and a nonzero status block
SYS$WAITFR	Waits for an event flag to be set
SYS$WFLOR	Waits for one of a number of event flags to be set
SYS$WFLAND	Waits for a number of event flags to be set

To refer to an event flag, use a number in one of the appropriate clusters. For example, to refer to a local event flag, you can use any number between 0 and 63 (excluding 24 through 31). To prevent accidental use of an event flag that is already in use elsewhere in your program, use the LIB$GET_EF and LIB$FREE_EF Run-Time Library routines. The LIB$GET_EF routine returns the number of a free local event flag; the LIB$FREE_EF returns a specified local event flag number to the list of free event flags. LIB$GET_EF and LIB$FREE_EF can only keep track of the event flags that they allocate and deallocate; therefore, to be effective, they must be used throughout your program. No similar routines exist for common event flags.

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). Unless you are passing the event flag to a routine that clears it for you, you should clear the event flag before using it. (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))
```

3.3.1.2 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.

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

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))
                    .
                    .
                    .
```

By default, a cluster name and common event flag cluster are disassociated when the image that associated them exits. When the last image associated with a cluster is disassociated, the common event flag cluster is deleted. Clusters that are deleted after all images are disassociated are called temporary clusters.

If you have PRMCEB privilege, you can create a permanent event flag cluster (set the **perm** argument to one 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 disassociated, the cluster is deleted.

3.3.1.3 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 checking that the system service has completed.

Typically, you pass an asynchronous system service an event flag and an I/O status block. 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.

3.3.2 Interrupting Execution with an AST

You can have the system interrupt execution of your program and transfer control to a subprogram by "queuing" an asynchronous system trap (AST). You queue an AST by specifying the name of the subprogram (the AST routine) as the **astadr** argument of certain system services. 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 FORTRAN I/O using SYS$INPUT or SYS$OUTPUT (for example, by specifying UNIT = *), you may receive a redundant I/O error. You must establish another channel to the terminal by explicitly opening the terminal (or using the Run-Time Library SMG$ routines).
- Shared routines—An AST routine may 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 may 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 services allow you to specify an AST routine to be delivered when the system service completes.

LIB$SPAWN	SYS$ENQ	SYS$ENQW	SYS$GETDVI
SYS$GETDVIW	SYS$GETJPI	SYS$GETJPIW	SYS$GETSYI
SYS$GETSYIW	SYS$QIO	SYS$QIOW	SYS$UPDSEC

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.

  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 not delivered; it 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 8.3 for more information and examples.

3.4 Interprocess Communication

Symbols, logical names, mailboxes, installed common blocks, and global sections allow you to pass information between images executing in different processes. In general, logical names are used to pass brief messages from one image to another. Mailboxes, installed common blocks, and global sections are used to carry on a dialog between images. The longer the messages in the dialog, the more reasonable it is to use installed common blocks or global sections.

3.4.1 Symbols and Logical Names

The following system-defined routines allow you to create, examine, and delete symbols and logical names.

LIB$SET_SYMBOL	Defines or redefines a symbol
LIB$GET_SYMBOL	Returns a symbol value
LIB$DELETE_SYMBOL	Deletes a symbol
SYS$CRELNM	Creates a logical name

SYS$CRELNT	Creates a logical name table
SYS$TRNLNM	Returns a logical name translation
SYS$DELLNM	Deletes a logical name or logical name table

Typically, you use logical names to pass information between two processes. If both processes are part of the same job, you can place the logical name in the process logical name table (LNM$PROCESS) or the job logical name table (LNM$JOB). (Note that if a subprocess is prevented from inheriting the process logical name table, you would have to 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.

The following program 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, lnm$job is not.)

```
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$_STRING
                 .
                 . ! Set REPETITIONS_STR
                 .
! Set up and create logical name REP_NUMBER in job table
LNMLIST(1).BUFLEN     = 3
LNMLIST(1).CODE       = LNM$_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$TRNLNM
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Declare itmlst
RECORD /ITMLST/ LNMLIST (2)
! Define item code
EXTERNAL LNM$_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
```

3.4.2 Mailboxes

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. Typically, when you invoke SYS$CREMBX, you specify two arguments.

- I/O channel (argument 2)—Specify an INTEGER*2 variable to receive the I/O channel number. This argument is required.
- Logical name (argument 7)—Specify the logical name to be associated with the mailbox. The logical name identifies the mailbox for other processes and for FORTRAN I/O 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.

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.

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 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 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
```

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 FORTRAN I/O. This is the recommended method of addressing a mailbox.
- Immediate—Read or write to a mailbox. Continue program execution after the operation completes. Use the SYS$QIOW system service.
- Asynchronous—Queue a read or write request to a mailbox. Continue program execution while the request executes. Use the SYS$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.

The *VAX/VMS System Services Reference Manual* describes the SYS$QIO and SYS$QIOW system services. It is recommended 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 *VAX/VMS I/O Reference Volume* for a description of the appropriate status block.

3.4.2.1 Synchronous Mailbox I/O

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

To open a mailbox for 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.

The following program segment opens a mailbox for the first time.

```
! 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')
```

Once a mailbox is open, you can use FORTRAN formatted, sequential READ and WRITE statements for I/O. FORTRAN automatically synchronizes I/O by not allowing an image to complete an I/O operation until a cooperating image has performed the opposite operation. For example, if an image performs a mailbox read operation, control is not returned to that image until a cooperating image performs a write operation to the same mailbox.

In the following example, one image passes another device names. 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.)

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
```

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
```

3.4.2.2 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.

To queue an immediate I/O request, invoke the SYS$QIOW system service with the following arguments (for details, see the *VAX/VMS System Services Reference Manual*).

- I/O channel (argument 2)—Use the channel returned by SYS$ASSIGN. The argument must be a word passed by value (%VAL).
- Function code and modifiers (argument 3)—Specify the function as one of the following (without the IO$M_NOW modifier the SYS$QIOW system service performs synchronous I/O):

IO$_WRITEVBLK .OR. IO$M_NOW	Write message
IO$_READVBLK .OR. IO$M_NOW	Read message
IO$_WRITEOF .OR. IO$M_NOW	Write end-of-file message

  The symbols are defined in the $IODEF module of the system object or shareable image library, and the FORTRAN definition library. The argument must be passed by value (%VAL).
- Status block (argument 4)—Define a status block of two words and a longword: the return status, the message size, and the process identification (PID) of the sender (for read) or receiver (for write). When reading data from a mailbox, always reference the data as a substring using the message size returned in the status block. If you reference the entire buffer, your data will include any excess characters from a previous operation using the buffer.
- User buffer (argument 7, or P1)—Define the user buffer as a character variable but pass it to SYS$QIOW by reference (%REF). SYS$QIOW ignores this argument when writing an end-of-file message.
- User buffer size (argument 8, or P2)—Specify the number of characters defined for the user buffer. For example, if you specify USER_BUFFER as CHARACTER*132, give USER_BUFFER_SIZE a value of 132. The argument must be passed by value (%VAL). SYS$QIOW ignores this argument when writing an end-of-file message.

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 the following example, 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, stopping 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.)

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$QIOW
! Create the mailbox
STATUS = SYS$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
```

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))
! 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
                             .
                             .
                             .
```

3.4.2.3 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 that you write, either (1) do not exit until 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$QIO system service as shown in Section 3.4.2.2; however, when specifying the function codes, do not specify the IO$M_NOW modifier. The *VAX/VMS System Services Reference Manual* describes the SYS$QIO system service; the I/O function codes are described in the *VAX/VMS I/O Reference Volume*. The SYS$QIO system service allows you to specify an AST to be executed and/or an event flag to be set when the I/O operation completes.

The following example 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 3.4.3.1 describes installed common blocks).

INSTALLED.FOR

```
! Installed common 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
! 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))
! 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
```

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
```

3.4.3 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.

3.4.3.1 Installed Common Blocks

To communicate between 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 shareable 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 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 *VAX/VMS Install Reference Manual*.)

   ```
   $ SET PROCESS/PRIVILEGE=CMKRNL
   $ INSTALL
   INSTALL> CREATE DISK$USER:[INCOME.DEV]INC_COMMON-
   _> /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.

Use 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. In the simplest case, one event flag in a common event flag cluster might indicate that a program is writing data and a second event flag in the cluster might indicate that a program is reading data. Before accessing the shared data, a program would examine the common event flag cluster to ensure that accessing the data would not conflict with an operation already in progress. Section 3.3.1 discusses program synchronization.

3.4.3.2 Global Sections

To share data using global sections, each process that plans to access the data includes a common block, which contains the variables for the data, of the same name. 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 9.2 for instructions.)

To create a global section, invoke SYS$CRMPSC as described in Section 9.2, adding 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 *VAX/VMS System Services Reference Manual* describes SYS$MGBLSC and its arguments. Typically, you specify only the following arguments:

- Section to map (argument 1)—Define the first argument as an integer array of two elements. Specify the location of the first variable in the common block as the value of the first array element 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, use the first element of the array or string; if the last variable in the common block is an array or string, use the last element of the array or string.)

  As described in Section 9.2, the program variables for the data are in a common block. Page align the common block at link time by specifying an options file containing the following link option (**name** is the name of the common block).

  ```
  PSECT_ATTR = name, PAGE
  ```

  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 odd page breaks in virtual memory.

- Section mapped (argument 2)—Define the second argument as an integer array of two elements. The value returned in the first array element should be the same as the address passed in the first element of the **inadr** argument. The value returned in the second element should be equal to or slightly more than (within 512 bytes, 1 block) the value passed in the second element of the **inadr** argument.
- Options (argument 4)—Specify SEC$V_GBL to indicate that the section is a global section.
- Section name (argument 5)—Specify a character string containing the name of the global section that you are mapping.

In the following example, one image, DEVICE.FOR, passes another image, GETDEVINF.FOR, device names. 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 9.2 if you have questions about either of these programs.

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
! 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$SIGNAL(%VAL(STATUS))
! Create the subprocess
STATUS = SYS$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$SIGNAL (%VAL(STATUS))
! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = %LOC (TERMINAL)
! Set write flag
SEC_MASK = SEC$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.

3.5 Intraprocess Communication

Per-process common blocks, local event flags, and symbols allow you to pass data from the current image to an image executed later by the same process. Logical names can be used for intraprocess communication; however, be sure to use supervisor-mode logical names. By default, SYS$CRELNM creates user-mode logical names, which are deleted when the creating image exits. Since per-process common blocks are least likely to be corrupted by images that execute between the time one image deposits information and another image reads it, they are recommended.

The Run-Time Library procedures LIB$RUN_PROGRAM and LIB$DO_COMMAND allow you to invoke the next image from the current image. Per-process common blocks, local event flags, symbols, and supervisor-mode logical names are equally acceptable for passing information between "chained" images since the image reading the information executes immediately after the image that deposited it.

3.5.1 Per-Process Common Blocks

To pass data from the current image to a later executing image, use a per-process common block (252 bytes long). The Run-Time Library procedure LIB$PUT_COMMON writes information to this common block; the Run-Time Library procedure LIB$GET_COMMON reads information from this common block.

The per-process common block is automatically created for you; no special declaration is necessary. To pass more than 252 bytes of data, put the data in a file and use the per-process common block to pass the name of the file.

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 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
                    .
                    .
                    .
```

Data in the per-process common block cannot be deleted or modified unless LIB$PUT_COMMON is invoked. Therefore, any number of images may be executed between INCOME.FOR and REPORT.FOR and, provided that LIB$PUT_COMMON has not been invoked, REPORT.FOR will read the correct data. Invoking LIB$GET_COMMON to read the per-process common block does not modify the data.

Although the descriptions of LIB$PUT_COMMON and LIB$GET_COMMON in the *VAX/VMS Run-Time Library Routines Reference Manual* specify a character string for the argument containing the data written to or read from the per-process common block, you can specify other types of data. However, since the data must be passed by descriptor, you must use the built-in function %DESCR to pass noncharacter data.

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 per-process common block is recommended since it uses fewer system resources than the permanent structures and does not require privilege.
(You need PRMMBX to create a permanent mailbox and PRMGBL to create a permanent global section.)

3.5.2 Passing Control

The Run-Time Library procedures 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.

- 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 run-time library 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 shown above.

  ```
                .
                .
                .
  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
  ```

- 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
```

3.6 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)

3.6.1 Requesting a Network Connection

To request a network connection, use the OPEN statement. The value of the FILE specifier must be a network task specification of the 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 *VAX/VMS Authorize Reference 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 and/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 8.3.4.)

```
! 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')
                            .
                            .
                            .
```

3.6.2 Completing a Network Connection

To complete a network connection, the program that is invoked by the command procedure named in the connection request uses the OPEN statement with a FILE specifier value of 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 or not the connection completes successfully.

3.6.3 Exchanging Messages

To exchange messages, cooperating programs can use FORTRAN READ and WRITE statements. In the following example, 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.

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$SIGNAL (%VAL(STATUS))
! Get user name on remote system
STATUS = LIB$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
```

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
```

3.6.3.1 Terminating a Network Connection

To terminate a network connection, use the CLOSE statement. 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 Implementation Cycle

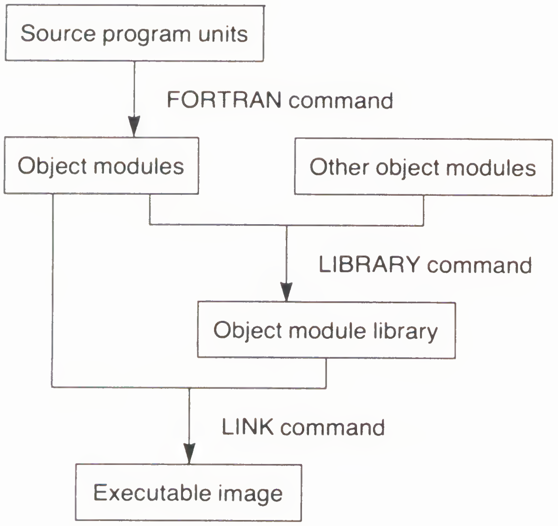
The term implementation cycle referes to the mechanics of processing your source program, subroutines, and functions into an executable program. The tools used in this process are the compiler, the linker, the librarian, and the debugger. This chapter discusses the FORTRAN implementation cycle; other languages follow a similar cycle. See the language-specific programming manual for more information about the language you are using.

Generally, you use the EDT editor to create and modify the files containing your source program units. For details about EDT, see the *VAX EDT Reference Manual*.

4.1 Compiling and Linking Programs

The general cycle for implementing an executable program (also called an executable image, application system, or subsystem) starts with compilation of the source program units into object modules and inclusion of those modules (except for the main program module) in an object library, then proceeds to linking the object modules into an executable image. (You are not required to put object modules in a library. You can link them directly.)

The following figure illustrates the FORTRAN implementation cycle. In addition to modules produced by the FORTRAN compiler, the object modules might include command language descriptions (see Chapter 7), error descriptions (see Chapter 10), symbol definitions (see Section 4.2), system-defined procedures (see Chapter 1), and other existing code that can be used by your system.



ZK-2077-84

4.1.1 Creating Source Programs

To create FORTRAN source programs (files with a file type of FOR) follow these rules:

- Statements—Start each new FORTRAN statement after the first tab on a new line (press RETURN and TAB)—otherwise an error results. In EDT, do not change the setting of the first tab (EDT changes a tab to multiple spaces if the tab is not a multiple of 8).
- Labels—Put a label (for example, for a FORMAT statement) in the first position of a new line, then press the TAB key.
- Spaces—Use spaces for legibility. They do not affect the compilation of the program into object code (provided that the spaces follow the first tab). For example, you can indent code on a line or insert spaces between words, numbers, and symbols—the code DOWHILE(STATUS) means the same as DO WHILE (STATUS).
- Continuation lines—You can continue a statement over any number of lines. Start each continuation line (after the tab) with any digit except 0. Do not extend lines beyond position 80.

- Comments—You can add comments to your code by (1) putting an asterisk, the letter C, or an exclamation point in the first position of a line (before the tab), (2) leaving a line all blank, (3) placing an exclamation point on a line (after the tab)—everything after the exclamation point is taken as a comment. In general, use the exclamation point and blank lines to describe your code, and the asterisk or letter C to leave code visibly intact but prevent it from being compiled.

The following FORTRAN example shows the contents of a file named GETFROM.FOR.

```
<---first tab

SUBROUTINE GET_FROM_BUFF (BUF,
2                         PTR,
2                         SIZ,
2                         CURLIN,
2                         LINE,     ! Return value
2                         LINE_LEN) ! Return value
! Gets one line from a buffer
! Declare arguments
CHARACTER BUF (204800)  ! Buffer
INTEGER PTR (10240),    ! Line pointers
2       SIZ (10240),    ! Line sizes
2       CURLIN          ! Line number of line being extracted
CHARACTER LINE (80)     ! Line being extracted
INTEGER LINE_LEN        ! Length of line being extracted
! Declare local data
INTEGER OFF,            ! Offset into buffer
2       I               ! For extracting a line character by character
! Get line from buffer
OFF = PTR (CURLIN) - 1
LINE_LEN = SIZ (CURLIN)
DO I = 1, LINE_LEN
  LINE (I) = BUF (OFF+I)
END DO
END                     ! Of subroutine
```

All the code above starts after the first tab. Note the use of spaces, blank lines, and exclamation points to format and add comments to the code. The lines beginning with digits are continuation lines—remember not to use a 0 as a continuation digit.

The code fragment below demonstrates the use of the first position of a source line.

```
<---first position

        <---first tab

        ! Declare I/O format
1       FORMAT (Q, INPUT_LINE)
        ! Establish special condition handler
C       CALL LIB$ESTABLISH (IO_HANDLER)
```

The number of the format statement must go in position 1. The C in position 1 of the CALL statement denotes a comment statement and is not compiled or executed (an asterisk or exclamation point also works)—at some future time, you could activate this statement by removing the C.

4.1.2 Compiling Programs

The FORTRAN command compiles source program units into object modules. In its simplest form, you need only name the file containing the source program unit or units. File types default to FOR for source files and OBJ for object files. The following command compiles the program unit or units in the file named REPORT.FOR in your default directory and produces a file named REPORT.OBJ which contains the resultant object module or modules.

```
$ FORTRAN REPORT
```

If REPORT.FOR contains one program unit named WRITE_REPORT, then REPORT.OBJ contains one object module named WRITE_REPORT. If REPORT.FOR contains two program units named WRITE_REPORT and CALC_SUMMARIES, then REPORT.OBJ contains two object modules with those names. The program unit names and file names are separate entities.

You can compile more than one source file at once. If you separate the names of the files with commas, the resultant object modules are placed in separate object files with names the same as the source files. If you separate the file names with plus signs, the resultant object modules are placed in one object file with the same name as the first source file.

Examples

```
$ FORTRAN REPORT,CALC
```

This example produces two object files named REPORT.OBJ and CALC.OBJ.

```
$ FORTRAN REPORT+CALC
```

This example produces one object file named REPORT.OBJ.

4.1.2.1 Development Systems

If you are compiling FORTRAN program units for a system under development, you should include the following qualifiers:

- /DEBUG—Appends to each object module information on local symbols, the entry point name (the name of the program or subprogram, which points at the first line of executable code), and line numbers (associating the line numbers with the code they generate). When you run the program, this information enables the debugger to communicate with you in terms of the program and subprogram names, variable names, and line numbers. It is recommended that you use the /NOOPTIMIZE qualifier with /DEBUG.
- /CHECK—Appends to each object module information to trap out-of-bounds subscripts, arithmetic overflows, and arithmetic underflows.
- /NOOPTIMIZE—Prevents optimization of the code for performance purposes. Optimization usually alters object modules so that they no longer exactly reflect the source program unit as you wrote it. In particular, optimization uses processor registers rather than your variables for some mathematical operations so that examination of those variables at run time does not yield true results.
- /LIST (optional)—Writes a listing to a file with the same name as the source file (if you do not specify otherwise) and a file type of LIS. You can print the file or display it. The listing includes the source statements in each program unit with their line numbers, and the data types and addresses (relative to the beginning of the program unit) of the variables. The listing is helpful in debugging. You may want to delete the listing after you finish using it because it requires a large amount of secondary storage. You can generate a listing without producing an object module by specifying the /LIST and /NOOBJECT qualifiers with the FORTRAN command.

The following FORTRAN example compiles the program unit in the file REPORT.FOR for development purposes. The compilation produces an unoptimized object module with full debugging information in a file named REPORT.OBJ and a listing in a file named REPORT.LIS.

```
$ FORTRAN/DEBUG/CHECK/NOOPTIMIZATION/LIST REPORT
```

4.1.2.2 Production Systems

When your program is stable and you wish to produce object modules for day-to-day use, you should compile the source files without the /DEBUG and /CHECK qualifiers (you no longer need the appended information) and without the /NOOPTIMIZATION qualifier (you want to optimize performance). Include the /LIST qualifier if you need a listing. The following example demonstrates the compilation of a production program unit.

```
$ FORTRAN/LIST REPORT
```

4.1.2.3 Errors

Your program will report any syntax errors at the end of the compilation. In FORTRAN, these messages start with the characters %FORT. Included in the message are the line number of the statement containing the error and the statement text containing the error. The messages also appear in the program listing.

Often the error message does not accurately state the problem; however, it usually locates the error correctly. In most cases, the errors are simple violations of the syntax rules or typographical errors. Common errors found in FORTRAN programs include the following:

- No parenthesis—Omitting the parenthesis at the end of an argument list generates the error MISSDEL (missing operator or delimiter symbol).
- No continuation digit—Omitting or mistyping the continuation digit, when a line is continued, generates various and numerous messages depending on the context, as FORTRAN attempts to interpret the continuation line as a new statement. Remember that 0 does not work as a continuation digit. Typing the continuation digit when it is not needed, extending a line beyond position 80, or using spaces in place of the first tab also gives unspecified results.
- No comma—Omitting a comma in a list of operands usually generates the error EXTCHAFOL (extra characters following a valid statement).
- No comment character—A full comment line must begin (before the tab) with an asterisk, the letter C, or an exclamation point. Omission of the comment character usually generates the error MISSDEL.
- Incorrect condition—The condition in an IF or DO WHILE statement must be enclosed in parentheses. The condition symbol must be of the form .EQ., .LE., and so on, not an equal sign or less-than sign. Each opening parenthesis must have a corresponding closing parenthesis. Violation of these rules usually generates the errors MISSDEL, NOPATH (no path to this statement), and INVCONSTR (invalid control statement using ELSE IF, ELSE, or END IF).

- No THEN on block IF—Omitting the word THEN in a block IF statement usually generates the errors MISSDEL, NOPATH, and INVCONSTR.
- No END for an IF or DO—Omitting the END IF statement on a block IF or the END DO statement on a DO statement results in the error OPEDOLOOP (unclosed DO loop or IF block).

A simple way to examine your compilation errors along with the source code causing the errors is to read the listing file with EDT (EDIT/READ) searching for each occurrence of %FORT. After determining the problem and the fix, edit your FORTRAN source file (not the listing file) to make the correction and recompile. If the errors are numerous or complex, print the listing.

An error that FORTRAN does not detect and for which you should always check is the incorrect specification of variable names. When FORTRAN encounters a name not explicitly declared, it creates a new variable implicitly, giving it a type based on the first letter in the name. In the following example, CURLINE is not treated as an error but as a new variable.

```
INTEGER CURLIN ! Line number of line being extracted
        .
        .
        .
OFF = PTR (CURLIN) - 1
LINE_LEN = SIZ (CURLINE) <--misspelling
DO I = 1, LINE_LEN
  LINE (I) = BUF (OFF+I)
END DO
```

Another simple way to specify a variable name incorrectly (besides misspelling it) is to omit required punctuation. The following example omits a comma in the argument list, causing FORTRAN to concatenate CURLIN and LINE into a new variable named CURLINLINE (remember that comments and spaces do not serve as delimiters in FORTRAN).

```
INTEGER FUNCTION GET_FROM_BUFF (BUF,      ! Passed
2                               PTR,      ! Passed
3                               SIZ,      ! Passed
4             missing comma-->  CURLIN    ! Passed
5                               LINE,     ! Returned
6                               LINE_LEN) ! Returned
```

In FORTRAN, you can force the explicit declaration of variable names by typing the statement IMPLICIT NONE in the definition part of your program unit. The errors shown above would then be reported at compilation time.

If you do not use the IMPLICIT NONE statement, you should make a habit of examining the sections of the listing near the end entitled VARIABLES and ARRAYS for unexpected entries. These sections list the variables as actually defined by FORTRAN.

4.1.3 Object Libraries

In any large development effort, you should store the object modules for your subprograms in an object library. An object library not only reduces the number of files you must maintain but greatly simplifies the linking process. To initially create an object library, type 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
```

To add or replace modules in a library, type the LIBRARY command with the /REPLACE qualifier followed by the name of the library (first parameter) and the name(s) of the file(s) 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 module(s) 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 recreate 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.

4.1.4 Linking and Executing Programs

The LINK command combines your object modules into one executable image, resolving global symbols such as entry point names, the names of common areas, and other variables declared as external. (Section 4.2 discusses symbols and the resolution of global symbols in more detail.) You can specify the object modules by naming the files or by naming the libraries in which they exist; the name of a library must be followed by the /LIBRARY qualifier. Typically, you name the file containing the main program first followed by the name of the library containing the subprograms. For example, if the file INCOME.OBJ contains your main program and the library file INCOME.OLB contains your subprograms, you specify the LINK command as follows:

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

For development systems, you normally include the /DEBUG qualifier. This qualifier appends to the image all the symbol and line number information appended to the object modules plus information on global symbols, and forces the image to run under debugger control when it is executed. The following example links INCOME and its subprograms for debugging.

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

If you invoke the image with the RUN command, you can inhibit the debugger by specifying RUN/NODEBUG. If you invoke the image through its own command, you must relink without the /DEBUG qualifier to inhibit the debugger. If you are executing a program without the debugger, you can invoke the debugger by pressing CTRL/Y and entering the word DEBUG as a command.

```
$ RUN INCOME
      .
      .
      .
CTRL/Y
$ DEBUG
```

For production systems, you not only want to inhibit the debugger (by not specifying /DEBUG) but you also want to inhibit the display of traceback information by specifying /NOTRACEBACK. The traceback information tells you on what lines and in what modules a run-time error occurs. However, this information should not be necessary after development and is rather messy in instances where your image simply continues execution after the issuance of a message (you want the message displayed but not the traceback information). The following example links INCOME and its subprograms for production.

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

If you set up your image as a DCL command through the use of command language descriptions (as explained in Chapter 7), execution of the image requires only that you enter the name of the command, followed by any parameters and qualifiers. In the following example, INCOME.EXE is executed by invoking the INCOME command with the /ENTER qualifier.

```
$ INCOME/ENTER
```

If the image is not set up as a DCL command, you can execute it by naming it as the parameter to the RUN command, as demonstrated below:

```
$ RUN INCOME
```

4.1.5 Privileged Programs

If a user needs privileges in order to execute your program, you may want to install the program as a privileged image. When a program is installed as a privileged image, the program has specified privileges, 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.

To install a FORTRAN program as a privileged image:

1. Use the DCL command SET PROCESS/PRIVILEGE=CMKRNL to give yourself CMKRNL privilege (required for use of the Install Utility).
2. Type INSTALL to invoke the interactive Install Utility.
3. When the INSTALL> prompt appears, type CREATE, the complete file specification of the file containing the executable program (file type defaults to EXE), and the /PRIVILEGED=(priv) qualifier, where **priv** is a list of the privileges that the program requires.
4. Press RETURN. The Install Utility installs your program as a privileged image and reissues the INSTALL> prompt.
5. Type EXIT to exit.
6. Use the DCL command SET PROCESS/PRIVILEGE=NOCMKRNL to remove CMKRNL privilege.

The following statements install $DISK1:[INCOME]GET_STATS as a privileged image with 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 type DELETE followed by the complete file specification of the image. Use the EXIT subcommand to exit. For complete documentation of the Install Utility, see the *VAX/VMS Install Reference Manual*.

4.1.6 Building Programs

You should take care to maintain your source, object, library, and image files so that they are all at the same level of development. For example, you do not want to make changes in a source program, produce a new image, and then lose the source program associated with the new image. In this regard, you should maintain a definitive set of source, object, library, and image files in their own directory.

You will save yourself a lot of time and trouble by writing command procedures to compile and link your modules and perform other development activities. Command procedures eliminate errors by ensuring that all necessary commands are entered correctly; they standardize your development procedures; and, of course, they simplify entry of the commands. However, develop your command procedures carefully and test them.

4.1.6.1 Build Process

Typically, you want to compile a program unit, enter the resultant object module in the library (if it is not the main program), and link it with other object modules in the executable program, purging old files and deleting unnecessary files. You want to ensure that the proper qualifiers are appended to the FORTRAN and LINK commands, depending on whether you are building a program for debugging, production, or with /TRACEBACK (without DEBUG). By using parameter 2 for control purposes, the following command procedure combines all of the above activities for an executable program called INCOME maintained in the directory $DISK1:[DEV.INCOME]. Alternatively, you could write separate command procedures to perform the separate types of builds (for example, DEBUG.COM, TRACE.COM, and PROD.COM).

$DISK1:[DEV.INCOME]INCOME.COM

```
$ ! INCOME.COM -- compiles and links a program unit for INCOME
$ ! Two parameters: P1 = file name of the program unit
$ !                 P2 = D -- debug (development system)
$ !                      T -- traceback (no debug)
$ !                      P -- production system
$ !
$ SET DEFAULT $DISK1:[DEV.INCOME]
$ ! Qualifiers for production system
$ QUALS = "/LIST"
$ ! Qualifiers for debug or traceback
$ IF (P2 .EQS. "D") .OR. (P2 .EQS. "T") THEN -
  QUALS = "/DEBUG/CHECK/NOOPTIMIZE/LIST"
$ ! Compile
$ FORTRAN'QUALS' 'P1'
$ ! If main module do not put in library
$ IF P1 .EQS. "INCOME" THEN GOTO LINK
$ ! Put subprogram module in library
$ LIBRARY/REPLACE INCOME 'P1'
$ DELETE 'P1'.OBJ;*
$LINK:
$ ! If linking for production system
$ IF P2 .EQS. "P" THEN LINK/NOTRACE
$ ! If linking for debug
$ IF P2 .EQS. "D" THEN LINK/DEBUG INCOME,INCOME/LIBRARY
$ ! If linking for traceback
$ IF P2 .EQS. "T" THEN LINK INCOME,INCOME/LIBRARY
$ ! Purge old files
$ PURGE
```

To build a new debugging system based on modifications to the program unit GET_STATS in the file GETSTATS.FOR, for example, you enter the following command:

```
$ @INCOME GETSTATS D
```

4.1.6.2 System Integrity

Once you have a program that works in your main build directory, you should avoid introducing changes that might make it not work. In this regard, a good practice when writing a new program unit or modifying an existing program unit is to build new executable programs (images) in another directory until you get a program that works properly. At this point, you can copy the new or modified source program unit to the build directory and build the new executable image there. (Do not just copy the image to the build directory. The image and the source files would then not match.) When you are building in a separate directory, you should still use the main build directory's copies of the other object modules. The following sequence of FORTRAN commands modifies the source file GETSTATS.FOR, builds new INCOME images in the [MY.INCOME] directory until a good image results, then copies GETSTATS.FOR to the main build directory [DEV.INCOME] and builds the image there.

```
$ SET DEFAULT $DISK1:[MY.INCOME]
$ COPY [DEV.INCOME]GETSTATS.FOR *
$ EDIT GETSTATS.FOR
        .
        .       ! Editing session
        .
$ @MYINCOME GETSTATS D ! See below for MYINCOME.COM
$ INCOME/ENTER
        .
        .       ! Debug session with INCOME.EXE
        .
(repeat the preceding steps until the image is correct)
$ COPY GETSTATS.FOR [DEV.INCOME]*
$ SET DEFAULT $DISK1:[DEV.INCOME]
$ @INCOME GETSTATS D
```

The command procedure MYINCOME.COM is similar to INCOME.COM but builds the image in [MY.INCOME] using the specified program unit (GET_STATS in this example) in [MY.INCOME] but obtaining the other object modules from [DEV.INCOME]. By default, the image created by the LINK command is given the file name of the first file named in the LINK command. Therefore, when you compile and link a library module from [MY.INCOME], you must use the /EXECUTABLE qualifier of the LINK command to explicitly name the new executable image [MY.INCOME]INCOME.

$DISK1:[MY.INCOME]MYINCOME.COM

```
$ ! MYINCOME.COM -- compiles and links a program unit in my area
$ ! Two parameters: P1 = file name of the program unit
$ !                 P2 = D -- debug (development system)
$ !                      T -- traceback (no debug)
$ !                      P -- production system
$ !
$ SET DEFAULT $DISK1:[MY.INCOME]
$ ! Qualifiers for production system
$ IF (P2 .NES. "P") THEN GOTO DEBUG
$ FORT_QUALS = "/LIST"
$ LINK_QUALS = "/NOTRACE"
$DEBUG:
$ ! Qualifiers for debug
$ IF (P2 .NES. "D") THEN GOTO TRACEBACK
$ FORT_QUALS = "/DEBUG/CHECK/NOOPTIMIZE/LIST"
$ LINK_QUALS = "/DEBUG"
$TRACEBACK:
$ ! Qualifiers for traceback
$ IF (P2 .NES. "T") THEN GOTO END_QUALS
$ FORT_QUALS = "/DEBUG/CHECK/NOOPTIMIZE/LIST"
$ LINK_QUALS = "/TRACE"
$END_QUALS:
$ ! Compile
$ FORTRAN'FORT_QUALS' 'P1'
$ ! Main program needs different link sequence
$ IF (P1 .EQS. "INCOME") THEN GOTO MAIN
$ ! Link subprogram
$ LINK/EXECUTABLE=[]INCOME'LINK_QUALS' -
  [DEV.INCOME]INCOME,-        ! Main program
  []'P1',-                    ! New program unit
  [DEV.INCOME]INCOME/LIBRARY ! Library
$ GOTO PURGE
$MAIN:
$ ! Link main program
$ LINK'LINK_QUALS' -
  INCOME,-                    ! New main program
  [DEV.INCOME]INCOME/LIBRARY ! Library
$PURGE:
$ ! Purge old files
$ PURGE
```

4.1.6.3 Checkin and Checkout

If more than one person is working on the development of a program, you must take precautions against destroying one another's changes. Suppose, for example, that Jane copies GETSTATS.FOR from the main build directory to her area and a few minutes later Dick copies the same program unit from the build directory to his area. They both make distinct modifications to their copies of the program unit, copy the program unit back to the main directory, and perform a build. The last person performing the build will destroy the other person's changes. You can resolve such conflicts verbally either by assigning ownership of each program unit to one person or telling one another what you are doing. However, automating the process so that only one person can work on a program unit at a time is much safer.

The following command procedure checks out a program unit from the main build directory by copying it to the current default directory and placing information in a special file in the build directory (CHECKOUT.DAT, which you must create as an empty file before using the command procedure) to indicate that the program unit is checked out. The command procedure aborts the checkout process if the program unit is already checked out.

```
$ ! CHECKOUT -- checks out a program unit
$ ! Two parameters -- P1 = file name of the program unit
$ !                   P2 = name of the user checking the file out
$ !
$ ON ERROR THEN EXIT ($STATUS)
$ !
$ ! Checkout file format:
$ ! unit-name user-name date
$ !
$ ! Read records from checkout.dat until program unit
$ ! is found or end of file occurs
$ OPEN CHECKOUT -                ! Open for read
  $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ MODULE = ""                    ! Initialize program unit name
$READLOOP:
$ READ/END_OF_FILE=ENDREADLOOP -
  CHECKOUT CHECKOUT_LINE
```

```
$ ! Extract program unit name and compare with parameter 1
$ MODULE_LEN = F$LOCATE (" ", CHECKOUT_LINE)
$ MODULE = F$EXTRACT (0, MODULE_LEN, CHECKOUT_LINE)
$ ! Drop out if parameter 1 is found
$ IF P1 .NES. MODULE THEN GOTO READLOOP
$ENDREADLOOP:
$ CLOSE CHECKOUT
$ !
$ ! If program unit already checked out, issue message
$ IF P1 .NES. MODULE THEN GOTO ENDALREADY
$ WRITE SYS$OUTPUT "Program unit already checked out"
$ WRITE SYS$OUTPUT CHECKOUT_LINE
$ EXIT
$ENDALREADY:
$ !
$ ! Otherwise check it out
$ ! Copy source file to default directory
$ COPY $DISK1:[DEV.INCOME]'P1'.FOR *
$ ! On error, close checkout files before exit
$ ON ERROR THEN GOTO EXIT
$ ! Copy checkout file to new file
$ OPEN OLDOUT $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ OPEN/WRITE NEWOUT $DISK1:[DEV.INCOME]CHECKOUT.DAT
$WRITELOOP:
$ READ/END_OF_FILE=ENDWRITELOOP OLDOUT CHECKOUT_LINE
$ WRITE NEWOUT CHECKOUT_LINE
$ GOTO WRITELOOP
$ENDWRITELOOP:
$ ! Add new record
$ CHECKOUT_LINE = P1 + " " + P2 + " " + F$TIME ()
$ WRITE NEWOUT CHECKOUT_LINE
$ ! Close files, purge old file, and exit
$EXIT:
$ STATUS = $STATUS
$ CLOSE OLDOUT
$ CLOSE NEWOUT
$ PURGE $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ EXIT (STATUS)
```

The following command procedure checks a program unit back into the main build directory by copying it from the default directory and deleting the associated entry in CHECKOUT.DAT. Exceptions are made if the program unit is checked out by another user (the checkin process is aborted) or the program unit is not checked out (you are asked if you want to proceed).

```
$ ! CHECKIN -- checks in a program unit
$ ! Two parameters -- P1 = file name of the program unit
$ !                   P2 = name of user checking the file in
$ !
$ ON ERROR THEN GOTO EXIT
$ FOUND = "FALSE"
$ !
$ ! Open checkout.dat for read and new one for write
$ OPEN OLDOUT $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ OPEN/WRITE NEWOUT $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ !
$ ! Read records to end of file
$UPLOOP:
$ READ/END_OF_FILE=ENDUPLOOP OLDOUT CHECKOUT_LINE
$ ! Look for program unit name
$ MODULE_LEN = F$LOCATE (" ", CHECKOUT_LINE)
$ MODULE = F$EXTRACT (0, MODULE_LEN, CHECKOUT_LINE)
$ ! If not program unit, write checkout record to new file
$ IF MODULE .EQS. P1 THEN GOTO USERNAME
$ WRITE NEWOUT CHECKOUT_LINE
$ GOTO UPLOOP
$ ! Otherwise program unit name was found
$USERNAME:
$ FOUND = "TRUE"
$ ! Look for user name
$ NAME = CHECKOUT_LINE - (MODULE + " ")
$ NAME_LEN = F$LOCATE (" ", NAME)
$ NAME = F$EXTRACT (0, NAME_LEN, NAME)
$ ! If correct program unit and user name,
$ ! copy source over and purge, and
$ ! delete checkout.dat record by not writing
$ IF NAME .NES. P2 THEN GOTO ANOTHER
$ COPY 'P1'.FOR $DISK1:[DEV.INCOME]'P1'.FOR
$ PURGE $DISK1:[DEV.INCOME]'P1'.FOR
$ GOTO UPLOOP
$ ! Otherwise write error message and copy record
$ANOTHER:
$ WRITE SYS$OUTPUT "Program unit checked out by another person"
$ WRITE SYS$OUTPUT CHECKOUT_LINE
$ WRITE NEWOUT CHECKOUT_LINE
$ GOTO UPLOOP
$ENDUPLOOP:
$ ! Reset status from EOF to success
$ $STATUS = 1
$ ! Inquire for copy if program unit was not found
$ IF FOUND THEN GOTO EXIT
$ INQUIRE YN "Program unit not checked out. Copy anyhow (Y or N)? "
$ IF YN THEN COPY 'P1'.FOR $DISK1:[DEV.INCOME]'P1'.FOR
$ !
$ ! Close checkout files, purge, and exit
$EXIT:
$ STATUS = $STATUS
$ CLOSE OLDOUT
$ CLOSE NEWOUT
$ PURGE $DISK1:[DEV.INCOME]CHECKOUT.DAT
$ EXIT (STATUS)
```

The CHECKIN.COM and CHECKOUT.COM command procedures are effective only if everyone working on the program uses them. No one must be allowed to shortcut the process by issuing COPY commands directly.

Unless all of the people working on the program have the same UIC, you must provide a means for the people working on the program to read from and write to the build directory. Several ways you can provide such access are as follows:

- Make the programmers members of one group and provide group members complete access to the build library.
- Use access control lists (ACLs) to provide all the programmers complete access to the build library.
- Set up a special account that the programmers must log into to use the checkin and checkout procedures.
- Provide the programmers with SYSPRV privilege. In the checkout procedure, use BACKUP instead of COPY to copy the source file to the build directory, and specify /OWNER=PARENT.

4.2 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. FORTRAN requires that you define all variables and code locations in terms of symbols, and also provides the capability of assigning symbols to constants. FORTRAN makes all the virtual address assignments—you cannot assign the actual addresses yourself, although you can assign contiguous addresses by placing variables in a record, array, or common block. 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.

4.2.1 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.) In FORTRAN programs, you define symbols in the following ways:

- Variables—The explicit or implicit declaration of a variable (as discussed in Chapter 6) allocates an area of virtual memory to hold the variable data and associates the area with a symbol.
- Blocks of variables—The declaration of a common block places variables in one physical area in virtual memory and associates the beginning of the area with a symbol.
- Constants—The PARAMETER statement assigns an immutable value to an area of virtual memory and associates the area with a symbol.
- Code—The PROGRAM, SUBROUTINE, or FUNCTION statement starts a block of code (the program or subprogram) and associates a symbol (the name of the program or subprogram) with the first executable statement in the block. If you do not start a program with a PROGRAM statement, the name of the program is **filename**$MAIN, where **filename** is the name of the file containing the program. The ENTRY statement also associates a symbol with an executable statement.

4.2.2 Local and Global Symbols

Symbols are either local or global in scope.

- Local symbols—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. In FORTRAN, the names of all variables and PARAMETER constants are local symbols.
- Global symbols—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. In FORTRAN, the names of programs, subprograms, entry points, and common blocks are global symbols.

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 (see Chapter 1) or data value (see the following paragraph)—you must define the symbol as external (EXTERNAL statement) or intrinsic (INTRINSIC statement; used for FORTRAN intrinsic functions and subroutines) in the definition portion of the program unit.

System facilities, such as the MESSAGE utility and the MACRO assembler, use global symbols to define data values. FORTRAN allows you to reference global data values defined by other programs; however, it does not allow you to define data values as global symbols.

4.2.3 Referencing Global Symbols

To reference a global symbol in FORTRAN, you must (1) define the symbol as external, and (2) reference the symbol as an argument to the %LOC built-in function. The following code fragment references the 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
```

Condition codes and other symbols defined in the system object and shareable image libraries are also defined in a FORTRAN specific library FORSYSDEF.TLB. Using the definitions from FORSYSDEF.TLB (see Section 4.2.5) allows you to reference condition codes and other system-defined symbols as local, rather than global, symbols.

4.2.4 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 will run 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).

4.2.4.1 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.

```
$ 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
```

4.2.4.2 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 Run-Time Library procedures 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.

4.2.4.3 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 LINK command. To name a single user library, equate the file name of the library to the logical name LNK$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
```

Additional user default libraries can be defined as LNK$LIBRARY_1, LNK$LIBRARY_2, and so on through LNK$LIBRARY_999. Do not skip any numbers.

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 example defines the default user library at the system level.

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

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.

4.2.4.4 Macro Libraries

Some system symbols are not defined in the system object and shareable image libraries. In such cases, the *VAX/VMS System Routines Reference Volume* notes that the symbols are defined in the system macro library and tells 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 similar to FORTRAN's PROGRAM statement; it is optional, but recommended.

```
<--first tab     <--second tab

.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 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
```

4.2.5 FORTRAN Definition Libraries and Files

The FORTRAN system definition library, SYS$LIBRARY:FORSYSDEF.TLB, defines the condition codes and other values defined in the system object and shareable image libraries (with the exception of those values specific to BASIC or COBOL) using data definition statements—mainly PARAMETER statements. Rather than accessing a symbol as a global symbol, you can include the FORSYSDEF symbol definition in your program unit by naming the library module that contains the symbol in an INCLUDE statement.

For example, the condition code RMS$_EOF is defined in the module $RMSDEF in the FORTRAN system definition library and the system object and shareable image libraries. To define the symbol for use in your program unit, specify the following INCLUDE statement in the definition part of your module (the INCLUDE specifier must be specified as a literal).

```
INCLUDE '($LIBDEF)'
```

The INCLUDE statement incorporates the data definition statements into your program unit so that the names of the values become local symbols. To reference the symbol, name the symbol; do not use %LOC. Note that the value definitions must be included in every program unit requiring them, not, for example, just in your main program.

The documentation identifies symbols defined in the system object and shareable image libraries (and in FORSYSDEF.TLB) by naming the defining module. If a symbol is defined only in the system macro library, you must use a MACRO routine (see Section 4.2.4.4) in order to reference the symbol.

You can create your own "global" values the same way FORSYSDEF.TLB does by placing PARAMETER statements, common blocks, or other data definition statements in text files and including them in your FORTRAN program units. If you have many INCLUDE modules, you should maintain them in a text library. The following example stores the parameter statements in SCREEN.TXT in an existing text library file named SYMBOLS.TLB (the file type defaults to TLB), and then deletes the source text file.

```
$ LIBRARY/TEXT/REPLACE SYMBOLS SCREEN
$ DELETE SCREEN.TXT;*
```

To include a module from a text library, place the name of the module in parentheses after the name of the file (the name of the text library defaults to SYS$SYSTEM:FORSYSDEF).

```
INCLUDE '$DISK1:[FORDEF]SYMBOLS(SCREEN)'
```

4.3 Shareable Images

A shareable image is a nonexecutable image which can be linked into 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 4.3.1), 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 4.3.4).

4.3.1 Creating Shareable Images

To create a shareable image follow these steps:

1. Compile the object module(s)—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 4.3.1.1 discusses transfer vectors); transfer vectors must be written in MACRO. The following template contains one transfer vector; repeat the three middle statements for each additional transfer vector.

```
<--- first tab
        <--- second tab
                <--- third tab

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

Naming the macro file with a name similar to that of the program unit's source file makes the transfer vector file easier to find. The MACRO and FORTRAN program units should not have identical names because after compiling the source files, you do not want the two object modules to have the same name. The following 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 following MACRO command produces an object module named X_GET_1_STAT in the file XGET1STAT.OBJ.

```
$ MACRO XGET1STAT
```

3 Write an options file—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 one. 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 4.3.1.2 for more information).

The following options file might be specified when you create the shareable image GET1STAT.

GET1STAT.OPT

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

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

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 module(s) 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
```

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 (see Section 4.3.2); 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. 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
```

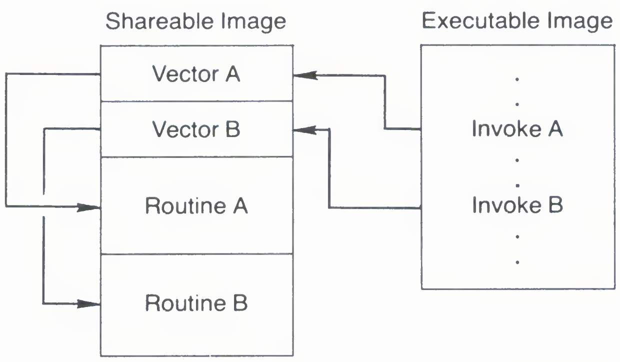
4.3.1.1 Transfer Vectors

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
```

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 (see the following figure). 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.



ZK-2078-84

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.

4.3.1.2 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 *VAX/VMS Linker Reference 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.

4.3.1.3 UNIVERSAL Options

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 *VAX/VMS Linker Reference 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 4.3.1.1), relinking is only necessary if the universal symbol is created using the UNIVERSAL option.

4.3.2 Shareable Image Libraries

To create a shareable image library, type the LIBRARY command with the /CREATE and /SHAREABLE qualifiers followed by the name of the library. The following command creates a shareable image library in a file named INCOMESHR.OLB (OLB is the default file type for both shareable image and object module libraries).

```
$ LIBRARY/CREATE/SHAREABLE INCOMESHR
```

To add or replace a shareable image in a shareable image library, type 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
```

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.

4.3.3 Linking Shareable Images

To specify a shareable image as input to the linker, you must specify either 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 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 which 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
```

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
```

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.

4.3.4 Shared Shareable Images

To allow executable images to share one copy of the shareable image in memory, install the shareable image as a shared image. When an executable image linked with a shared shareable image accesses the shareable image, if a copy of the image is already in memory, the executable image uses that copy. If no copy of the shared shareable image is in memory, the executable image copies the shared shareable image into memory. Unless the shareable image is likely to be accessed by more than one image at a time, do not bother to install the shareable image as a shared image.

To install an image as shared, follow the steps described in Section 4.1.5 for installing an image as privileged. However, instead of specifying the /PRIVILEGED qualifier, specify the /SHARED qualifier.

4.4 Listings

You can generate compiler listings and image maps during the compile and link phases of your program's development.

4.4.1 Compiler Listings

To generate a listing file, specify the /LIST qualifier when you enter the FORTRAN command interactively.

```
$ FORTRAN/LIST INCOME
```

If the program is compiled as a batch job, the listing file is created by default; specify the /NOLIST qualifier to suppress creation of the listing file. (In either case, the listing file is not automatically printed.) By default, the listing file has the same file name as that of the first source file specified with the FORTRAN command and a file type of LIS. You can include a file specification with the /LIST qualifier to override this default. For example, to generate a listing file with the specification INCOME1.LIS, specify

```
$ FORTRAN/LIST=INCOME1.LIS INCOME
```

The components of a listing depend upon the qualifiers specified with /LIST. By default, a listing contains the source program section, storage map, and compilation summary. You can additionally request optional source lines, machine code, and supplementary symbol information.

4.4.1.1 Source Program Listing

The source listing contains the source code plus line numbers generated by the compiler. The line numbers appear, one per line, in the left margin; they are the lines referred to by error messages and debugger displays and commands. If you created the source file with editor-generated line numbers, those line numbers appear to the left of the compiler-generated line numbers. FORTRAN error messages refer to editor line numbers when present; otherwise, they refer to the compiler-generated numbers. In the following excerpt from a listing of the function CALC_SUMS, only compiler-generated line numbers are present.

```
0001          INTEGER FUNCTION CALC_SUMS (TOTAL_HOUSES,
0002          2                            PERSONS_HOUSE,
0003          2                            ADULTS_HOUSE,
0004          2                            INCOME_HOUSE,
0005          2                            AVG_PERSONS_HOUSE,
0006          2                            AVG_ADULTS_HOUSE,
0007          2                            AVG_INCOME_HOUSE,
0008          2                            AVG_INCOME_PERSON,
0009          2                            MED_INCOME_HOUSE)
0010          ! Calculates averages and median from statistics
                                .
                                .
                                .
0050          ! Calculate median income per house
0051          J = 1
0052          DO I = 2, 101
0053            IF (MEDIAN (I) .GT. MEDIAN (J)) J = I
0054          END DO
0055          MED_INCOME_HOUSE = J * 1000
0056
0057          ! Return
0058          CALC_SUMS = STATUS_OK
0059          END  ! Of subroutine
```

To include lines of source code which are usually omitted from the compiler listing, use the /SHOW qualifier in addition to the /LIST qualifier. For example, to list the source lines of a file specified by the INCLUDE statement, specify /SHOW=INCLUDE. To list source lines generated by a preprocessor (for example, the VAX DBMS FORTRAN Data Manipulation Language), specify /SHOW=PREPROCESSOR.

4.4.1.2 Machine Code Listing

To include a listing of the compiler-generated object code, specify the /MACHINE_CODE qualifier with the /LIST qualifier.

```
$ FORTRAN/LIST/MACHINE_CODE INCOME
```

The first line of machine code contains a .TITLE directive that indicates the name of the program unit to which the machine code corresponds. If the program unit is the main program, the title is either the name declared in the PROGRAM statement or, if there is no PROGRAM statement, **filename**$MAIN. If the program unit is a BLOCK DATA subprogram, the title is either the name declared in the BLOCK DATA statement or, if there is no BLOCK DATA statement, **filename**$DATA. (In the previous statements, filename refers to the name of the source file.)

The lines following .TITLE contain information about storage, such as the variables initialized in data type statements. Following the storage information are machine instructions, represented by VAX MACRO mnemonics. Compiler-generated line numbers are listed in the right margin, and the virtual address of the beginning of each line is listed in the left margin. For example, line 53 of the source program unit CALC_SUMS translates into five lines of machine code.

```
                                                  ; 0053
00BB  MOVL  J(R11), R0
00BF  MOVL  I(R11), R1
00C3  CMPL  MEDIAN-4(R11)[R1], MEDIAN-4(R11)[R0]
00CC  BLEQ  L$6
00CE  MOVL  I(R11), J(R11)
```

The VAX general registers (0 through 12) are represented by R0 through R12. When register 12 is used as the argument pointer, it is represented as AP. The frame pointer (register 13) is represented as FP; the stack pointer (register 14) as SP; and the program counter (register 15) as PC. Integer constants are shown as signed integer values, real and complex constants as unsigned hexadecimal values preceded by ^X. Addresses are represented by the program section name plus the hexadecimal offset within that program section. Program sections are indicated by PSECT lines, and labels that the compiler generates for its own use appear as L$n.

The following is an excerpt of machine code generated for the program unit CALCSUMS.

```
      .TITLE  CALC_SUMS
      .IDENT  01
01AC  .PSECT  $LOCAL
01AC  .LONG    ^X040A0004
01B0  .LONG    ^X00000000
01B4  .LONG    ^X01E00000,^X00002000
                                  .
                                  .
                                  .
0000  .PSECT  $CODE
0000  CALC_SUMS::
0000  .WORD    ^M<IV,R11>
0002  MOVAL   $LOCAL+^X198, R11
0009  MOVL    PERSONS_HOUSE(AP), $LOCAL+^X1B0(R11)
000E  SUBL3   #4, PERSONS_HOUSE(AP), $LOCAL+^X1BC(R11)
0014  MOVL    ADULTS_HOUSE(AP), $LOCAL+^X1D0(R11)
0019  SUBL3   #4, ADULTS_HOUSE(AP), $LOCAL+^X1DC(R11)
001F  MOVL    INCOME_HOUSE(AP), $LOCAL+^X1F0(R11)
0024  SUBL3   #4, INCOME_HOUSE(AP), $LOCAL+^X1FC(R11)
                                                                    ; 0034
002A  MOVL    @TOTAL_HOUSES(AP), $LOCAL+^X20C(R11)
002F  MOVL    #1, I(R11)
0033  CMPL    I(R11), $LOCAL+^X20C(R11)
0038  BGTR    L$4
                                  .
                                  .
                                  .
                                                                    ; 0055
00DC  MULL3  #1000, J(R11), @MED_INCOME_HOUSE(AP)
                                                                    ; 0058
00E6  MOVL   #1, CALC_SUMS(R11)
                                                                    ; 0059
00EA  MOVL   CALC_SUMS(R11), R0
00EE  RET
     .END
```

4.4.1.3 Storage Map

The storage map portion of a compiler listing contains summary information on program sections, variables, and arrays. You can also request additional symbol information in certain sections of the storage map by specifying the /CROSS_REFERENCE qualifier with the /LIST qualifier (see the end of this section).

Note that data addresses are specified in hexadecimal as offsets from the start of a program section or the argument pointer (AP). Indirect addressing is indicated by an at sign (@) following an address field. (In indirect addressing, the address specified by the program section or AP, plus the offset, points to the address of the data—not to the data itself.)

The following is a list of the information which can be generated:

1 Program sections—Describe each program section, including its PSECT number, name, size, and attributes, as well as the total memory allocated for all program sections. For example, the following program section summary describes $CODE (the PSECT containing executable statements) as being 224 (decimal) bytes of code that is position-independent, concatenated, relocatable, local, shareable, executable, readable, nonwriteable, and longword aligned.

```
PROGRAM SECTIONS

Name   Bytes  Attributes

$CODE   224   PIC CON REL LCL   SHR   EXE RD NOWRT LONG
$LOCAL 524    PIC CON REL LCL NOSHR NOEXE RD   WRT LONG

Total Space Allocated  748
```

2 Entry points—List each entry point (an executable statement at which execution of a program unit can begin) and its address. If the program unit is a function, its data type is also listed. For example, the following summary indicates the program unit CALC_SUMS is a function with an INTEGER*4 data type.

```
ENTRY POINTS

Address     Type  Name

0-00000000  I*4  CALC_SUMS
```

3 Statement function summary—Lists the entry point, data type, and name of each statement function. If all of the references to a statement function generate in-line code, the body of the statement function is not compiled, and a double asterisk (**) appears instead of an address.

4 Variables—List the addresses, data types, and names of all variables except arrays. In the following example, each variable is a REAL*4 type.

```
VARIABLES
   Address      Type   Name

 2-0000019C     R*4   ADULTS
AP-00000018@    R*4   AVG_ADULTS_HOUSE
AP-0000001C@    R*4   AVG_INCOME_HOUSE
```

5 Arrays—List the address, data type, and name of each array, plus its total size (in bytes) and number of elements (listed as dimensions). If the array is an adjustable array or assumed-size array, its size is shown as double asterisks (**) and each adjustable dimension bound is shown as a single asterisk (*). In the following example, all arrays are one dimensional.

```
ARRAYS

Address      Type  Name           Bytes   Dimensions

AP-0000000C@  R*4  ADULTS_HOUSE    8192    (2048)
AP-00000010@  R*4  INCOME_HOUSE    8192    (2048)
 2-00000000   I*4  MEDIAN           404     (101)
AP-00000008@  R*4  PERSONS_HOUSE   8192    (2048)
```

6 Namelist Summary—Lists the name and address of each namelist.

7 Label Summary—Lists the number and address of each user-defined statement label. FORMAT statement labels are suffixed with an apostrophe (′). If the label address field contains double asterisks (**), the label was not used or referred to by the compiled code.

8 Functions and subroutines referenced—List all external symbols referenced by the source program (except references to symbols used as dummy arguments) and their data types. The following excerpt shows the external symbols referenced by INCOME.

```
FUNCTIONS AND SUBROUTINES REFERENCED

Type  Name                                 Type  Name

I*4  CLI$GET_VALUE                         I*4  REPORT
I*4  FIX_STATS                             I*4  CONVERT_FIXES
I*4  GET_STATS                                  FOR$OPEN
     INCOME__INSFIXVAL                          INCOME__FORIOERR
     LIB$SIGNAL                            I*4  LIB$GET_LUN
I*4  CLI$PRESENT
     FOR$CLOSE
     INCOME__BADFIXVAL
     INCOME__NOACTION
```

To generate additional symbol information in various sections of the storage map, specify the /CROSS_REFERENCE qualifier with the /LIST qualifier.

```
$ FORTRAN/LIST/CROSS_REFERENCE INCOME
```

The /CROSS_REFERENCE qualifier includes the following symbol information in the storage map: the lines where symbols are defined and initialized, the lines where the values of symbols are modified, the lines where symbols are actual arguments, and the number of times a symbol occurs in each line. For example, the following excerpt indicates that the variable ADULTS was referenced four times: in lines 24, 36 (where it was referenced twice and its value was modified), and 46.

Key to Reference Flags

=	Value modified
#	Defining reference
A	Actual argument, possibly modified
D	Data initialization
(n)	Number of occurrences on line

```
ENTRY POINTS
 Address    Type  Name                 References
0-00000000   I*4  CALC_SUMS             1        58=

VARIABLES
   Address   Type  Name                References
 2-0000019C   R*4  ADULTS              24      36(2)=   46
AP-00000018@ R*4  AVG_ADULTS_HOUSE     1      15       46=
AP-0000001C@ R*4  AVG_INCOME_HOUSE     1      15       47=
AP-00000020@ R*4  AVG_INCOME_PERSON    1      15       48=
AP-00000014@ R*4  AVG_PERSONS_HOUSE    1      15       45=
                                 .
                                 .
                                 .
```

4.4.1.4 Compilation Summary

The compilation summary lists the qualifiers used with the FORTRAN command and the compilation statistics.

```
COMMAND QUALIFIERS
FORTRAN /DEBUG/LIS/MACHINE_CODE CALCSUMS
 /CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
 /DEBUG=(SYMBOLS,TRACEBACK)
 /STANDARD=(NOSYNTAX,NOSOURCE_FORM)
 /SHOW=(NOPREPROCESSOR,NOINCLUDE,MAP)
 /F77  /NOG_FLOATING  /I4  /OPTIMIZE  /WARNINGS
 /NOD_LINES /NOCROSS_REFERENCE  /MACHINE_CODE
 /CONTINUATIONS=19
COMPILATION STATISTICS
Run Time:           1.29 seconds
Elapsed Time:       4.02 seconds
Page Faults:        163
Dynamic Memory:     160 pages
```

4.4.2 Image Maps

An image map contains information about the link process and the image it produces; you can use an image map to locate link-time errors, to study the layout of the image in virtual memory, to keep track of global symbols, and so on. When you link your program interactively, you must specify the /MAP qualifier with the LINK command to generate an image map file.

```
$ LINK/MAP INCOME
```

If the program is linked in a batch job, a map is generated by default. To suppress the creation of a map file, specify /NOMAP. (In either case, the image map is not automatically printed.) By default, the map file has a file type of MAP and the same file name as that of the first object file specified with the LINK command. To override this default, specify the file name with the LINK command. For example, to generate a map file with the specification INCOME1.MAP, specify

```
$ LINK/MAP=INCOME1.MAP INCOME
```

The components of a map file depend upon the qualifiers specified with the /MAP qualifier. If you specify only LINK/MAP, the image map contains five sections; if you specify LINK/MAP/FULL, it contains eight sections; if you specify LINK/MAP/BRIEF, it contains only three sections. The following table indicates which map sections are generated by each qualifier combination.

Qualifier	Image Map Sections
/MAP/BRIEF	Object module synopsis Image synopsis Link run statistics
/MAP	Object module synopsis Image synopsis Link run statistics Program section synopsis Symbols by name
/MAP/FULL	Object module synopsis Image synopsis Link run statistics Program section synopsis Symbols by name Symbols by value Module relocatable reference synopsis Image section synopsis

Not only does a full map (an image map generated by the /MAP/FULL qualifiers) contain more sections than a default or brief map does, but some of the full map's sections may also contain more information than the same sections in a default or brief map. This information concerns modules or shareable images that were implicitly included (not explicitly specified) in the link. The sections that contain more information when generated with the /FULL qualifier are: object module synopsis, program section synopsis, symbols by name, and symbol cross-reference.

With the /MAP and /MAP/FULL qualifiers you can also specify the /CROSS_REFERENCE qualifier, which replaces the symbols by name section with a symbol cross-reference section.

5 Using the Debugger

Once you have successfully compiled and linked your program, you can use the debugger to check it for errors while it executes. For a complete description of the debugger, see the *VAX/VMS Debugger Reference Manual.*

Debugging sessions differ depending on the size and content of your program. Assuming that you have compiled and linked your FORTRAN program with the debugger (FORTRAN/LIST/DEBUG/NOOPTIMIZE; LINK /DEBUG), the following commands are generally useful in locating an error.

- SET MODULE/ALL—Gives the debugger access to all program variables.
- SET EXCEPTION BREAK—Indicates that the debugger should take control when an error occurs.
- GO—Begins program execution. When an error occurs, the debugger displays the line at which it occurred, and then prompts you for a command.
- EXAMINE variable—Examines a local variable. When an error occurs, you should examine local variables to ensure that they all contain the expected values.
- EXIT—If you are still unsure of the problem, use the EXIT command to exit from the debugger and execute the program again. (You can reexecute a program without exiting, as shown in Section 5.4.1.1; however, variable values may be incorrect.)

For subsequent debugging sessions, use the following commands to watch program execution more carefully.

- SET BREAK routine—After using the SET MODULE/ALL command as described above, use the SET BREAK command to indicate that the debugger should take control when the routine that caused the error begins executing. (Specify the name of the offending routine as the parameter of the SET BREAK command.)
- SET MODE SCREEN—Enters screen mode in the debugger so that you can examine the source code as your program executes.
- STEP—Executes the current routine one line at a time. Use STEP to ensure that all statements are executing in the expected order.

If you still have not located the error, you might use tracepoints, as described in Section 5.4.2.1, to ensure that your routines are executing in the proper order.

5.1 Invoking and Terminating the Debugger

Normally, to invoke the debugger, you compile and link a program with the /DEBUG qualifier and then execute the program. When a program linked with the debugger executes, the debugger prompt appears and the program executes under debugger control. To terminate a debugging session, type the EXIT command or press CTRL/Z.

5.1.1 Invoking the Debugger

To make all symbol information available during a debugging session, specify the /DEBUG qualifier with both the FORTRAN and LINK commands. Subsequent execution of your program invokes the debugger. For example, to debug a FORTRAN program named INCOME.FOR, enter the following commands.

```
$ FORTRAN/DEBUG/NOOPTIMIZE INCOME
$ LINK/DEBUG INCOME
```

To invoke the debugger, issue the RUN command at DCL level. The debugger identifies itself as follows:

```
$ RUN INCOME
                    VAX DEBUG VERSION 4.4
%DEBUG-I-INITIAL, language is FORTRAN, module set to 'INCOME'
DBG>
```

The /NOOPTIMIZE qualifier of the FORTRAN command prevents the FORTRAN compiler from optimizing your program, thus ensuring that the executable image exactly reflects your source code. If you omit the /NOOPTIMIZE qualifier, examining variables while debugging may not produce valid results. For systems under development, you should also include the /CHECK and /LIST qualifiers with the FORTRAN command; see Section 4.1 for more information about these qualifiers.

If your program is running without debugger control and you want to invoke the debugger, use CTRL/Y to interrupt the execution of your program and then specify the DCL command DEBUG. The module and language information that exists in your program at the time of its interruption is preserved; however, the debugger will not have access to full symbol information.

5.1.2 Interrupting the Debugger

Once you have invoked the debugger, you can interrupt the currently executing source code by pressing CTRL/Y or entering the SPAWN command.

- CTRL/Y—Terminates any debugger command or source code that is executing and puts you at DCL command level. You can return to the debugger with either the DCL command CONTINUE or DEBUG, as long as while you were at DCL level you only executed commands performed within the command interpreter (see the *VAX/VMS DCL Dictionary* for a list of these commands.) The CONTINUE command continues the debugging session where you interrupted it. The DEBUG command aborts the interrupted command before continuing the debugging session (useful for aborting an infinite loop or long command).

- SPAWN—Creates a subprocess that allows you to use DCL commands without terminating the debugging session. If you specify a DCL command as a parameter of SPAWN, the spawned subprocess executes the command and immediately returns debugger control. The following example executes MAIL in the middle of a debugging session.

  ```
  DBG> SPAWN MAIL
       .
       .
       .
  MAIL> EXIT
  %DEBUG-I-RETURNED, control returned to process USER
  DBG>
  ```

  Specify the SPAWN command without a parameter to go to DCL command level and be able to enter any number of DCL commands. To resume your debugging session, specify the LOGOUT command.

  ```
  DBG> SPAWN
  $ MAIL
       .
       .
       .
  MAIL> EXIT
  $ LOGOUT
   Process USER_1 logged out at 28-JUL-1984  09:59:12:45
  %DEBUG-I-RETURNED, control returned to process USER
  DBG>
  ```

5.1.3 Terminating the Debugger

The following message indicates that your program has executed normally.

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

To terminate the debugging session, press CTRL/Z or type EXIT.

```
DBG> EXIT
$
```

The dollar sign prompt indicates that you are at DCL command level.

5.2 Entering Debugger Commands

To enter a debugger command, type it after the debugger prompt (DBG>) and press RETURN. A number of keypad keys are also defined commands. If you are in keypad mode (the default) and wish to execute one of these commands, press the appropriate keypad key.

5.2.1 Using Debugger HELP

To display the list of debugger topics on which information is available, type HELP. To display information about a particular topic, type HELP plus the topic name. For example, to display information about the use of the qualifier /ALL with the SET MODULE command, specify

```
DBG> HELP SET MODULE/ALL
```

5.2.2 Abbreviating Debugger Commands

Debugger commands, like DCL commands, can be abbreviated to unique characters. For example, the command CANCEL EXCEPTION BREAK could be entered as CAN EX BR. (For clarity, this document does not abbreviate commands.) In addition, you can abbreviate a debugger command by using the DEFINE command with its /COMMAND qualifier to equate the command to a shorter symbol name. The following example creates the symbol CEB to abbreviate the command CANCEL EXCEPTION BREAK.

```
DBG> DEFINE/COMMAND CEB = "CANCEL EXCEPTION BREAK/ALL"
```

To make your symbol definitions available at each debugging session, include them in a debug initialization command file that is executed at the outset of all debugging sessions (see Section 5.7.3).

5.2.3 Using the Keypad

Debugging in keypad mode allows you to use the keypad keys to enter debugger commands. To invoke keypad mode, specify

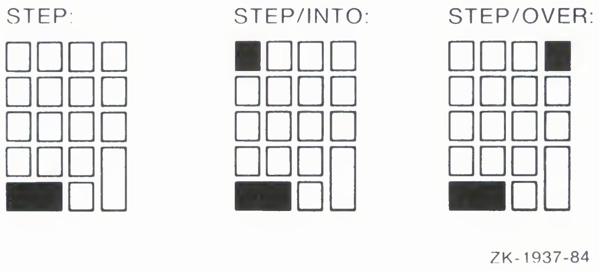
```
DBG> SET MODE KEYPAD
```

To terminate keypad mode, specify

```
DBG> SET MODE NOKEYPAD
```

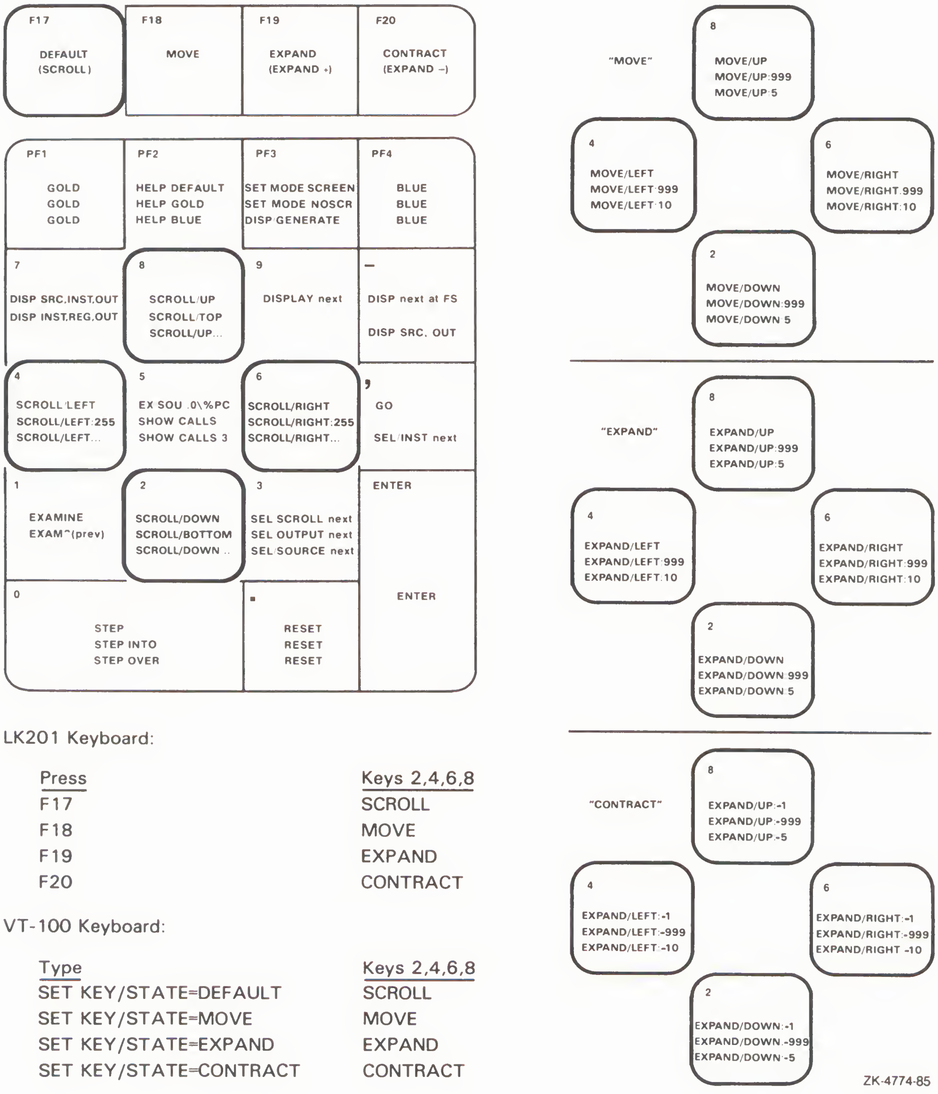
The debugger provides a set of default key definitions for each key in the keypad; you can redefine each of these keys with the DEFINE/KEY command. Each key in the default keypad can perform at least one debugging command; most of the keys perform three: one command is entered by pressing the keypad key, a second command is entered by pressing the PF1 key and then the keypad key, and a third command is entered by pressing the PF4 key and then the keypad key.

For example, the keypad key labeled 0 enters the STEP command by default; the 0 key in combination with the PF1 key enters the STEP/INTO command; and the 0 key in combination with the PF4 key enters the STEP/OVER command.



5.2.3.1 Default Key Definitions

The following figure shows the default definitions of each keypad key. (The commands are listed in the block designating the keypad key to which they are assigned.) The first command listed is the command you enter by pressing the single key, the second command listed is the command you enter by pressing the PF1 key in combination with the key, and the third command listed is the command you enter by pressing the PF4 key in combination with the key.



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

Some keys (such as PF3, SET MODE SCREEN) enter the command immediately when pressed. The keys whose commands are followed by an ellipsis (such as PF4 plus key 8, SCROLL/UP...) take parameters that you must type on the main keyboard; enter the completed command by pressing either RETURN or ENTER. For example, to enter the command SCROLL /UP..., first press PF4 and key 8. The command SCROLL/UP is echoed on the screen, you type the number of lines you would like to scroll, and then you press ENTER to execute the command.

5.2.3.2 User Key Definitions

The debugger DEFINE/KEY command (similar to the DCL DEFINE/KEY command) allows you to assign a debugger command to a keypad key. For example, to define the keypad key 7 to enter and execute the SET MODULE /ALL command, specify

```
DBG> DEFINE/KEY/TERMINATE KP7 "SET MODULE/ALL"
```

You must be in keypad mode to define, use, display, or delete a keypad key. To display the current definition of a keypad key, specify

```
DBG> SHOW KEY key
```

To delete a key's definition, specify

```
DBG> DELETE/KEY key
```

You can put key definitions in a debugger initialization file so that the key is available whenever the intialization file is executed (see Section 5.7.3).

5.3 Using Screen Displays

Screen mode debugging allows you to keep a variety of debugging information on the screen by dividing the screen into sections and displaying different types of information in each section. The sections of the screen are called windows, and the contents of the windows are called displays. In screen mode, the debugger defines a number of default windows and maintains five default displays: a display of source lines (SRC), a display of instruction lines (INST), a display of debugger output (OUT), a display of register contents (REG), and a display containing the debugger prompt and your input.

5.3.1 Invoking and Terminating Screen Mode

To invoke screen mode, press the keypad PF3 key or enter the SET MODE SCREEN command. Three displays appear on the screen by default: the default SRC display appears in window H1 (the top half of your screen), the default OUT display appears in window S45 (lines 13 through 20), and the PROMPT display appears in window S6 (lines 21-24). The screen output includes 24 lines; six sections containing four lines each.

The display name and characteristics are placed on the title line of the window. The following screen appears when you execute a SET MODE SCREEN command followed by a STEP command when debugging the program INCOME. The arrow in the left column indicates the current location of the program counter which points to the next statement to be executed. The STEP command brings you to the first executable statement of the program.

```
DBG> SET MODE SCREEN
DGB> STEP
DBG>
```

```
 --SRC: module INCOME---scroll-source-----------------------------------
   219:        ! Declare subprograms invoked as functions
   220:        2       CLI$PRESENT,
   221:        2       LIB$GET_LUN
   222:
   223:        ! Get logical unit number for STATS.SAV
-> 224:        STATUS = LIB$GET_LUN (STATS_LUN)
   225:        IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   226:
   227:        ! Get name of file containing data base
   228:        CALL CLI$GET_VALUE ('STATS_FILE',
   229:        2                   STATS_FILE,
--OUT---output---------------------------------------------------------
stepped to INCOME\%LINE 224
   224:        STATUS = LIB$GET_LUN (STATS_LUN)


--PROMPT -error-program-prompt-----------------------------------------
DBG>

-----------------------------------------------------------------------
```

By default, the SRC display points to the next executable source line and shows the five lines preceding and following it. The entire source program is available through scrolling, as long as the conditions for regular source display are met (see Section 5.4.3 for a list of these conditions). The OUT display shows debugger output, such as responses to SHOW and EXAMINE commands. (The 100 most recent lines of debugger output are available through scrolling.) The REG and INST displays are also available by specifying the DISPLAY REGISTER or DISPLAY INST commands; the

REG display shows the current contents of machine registers and the INST display shows the instruction code.

To display information about existing screen displays (including those not currently displayed on the screen), specify the SHOW DISPLAY command. The information is output to display OUT.

```
DBG> SHOW DISPLAY
DBG>
```

```
 --SRC: module INCOME---scroll-source----------------------------------
   219:        ! Declare subprograms invoked as functions
   220:        2       CLI$PRESENT,
   221:        2       LIB$GET_LUN
   222:
   223:        ! Get logical unit number for STATS.SAV
-> 224:        STATUS = LIB$GET_LUN (STATS_LUN)
   225:        IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   226:
   227:        ! Get name of file containing data base
   228:        CALL CLI$GET_VALUE ('STATS_FILE',
   229:        2                   STATS_FILE,
--OUT---out-----------------------------------------------------------
display SRC at H1, size = 64
    kind = SOURCE (EXAMINE/SOURCE .%SOURCE_SCOPE\%PC)
display INST at H1,size = 64, removed
    kind = INSTRUCTION (EXAMINE/INSTRUCTION .0\%PC)
display REG at RH1, size = 64, removed, kind = REGISTER
display OUT at S45, size = 100, kind = OUTPUT
display PROMPT at S6, size = 64, kind = PROGRAM
-PROMPT -error-program-prompt-----------------------------------------
DBG>

----------------------------------------------------------------------
```

When debugger output from a single command exceeds the dimensions of display OUT (the output from the SHOW WINDOW command, for example), the beginning of the display is not visible. To view the entire display, you can scroll the display (see Section 5.3.3.3), place the display into window FS (full screen) or another large screen region (Section 5.3.3.1), or terminate screen mode (SET MODE NOSCREEN) before entering the command. (A subsequent SET MODE SCREEN command restores the screen displays.)

5.3.2 Defining Windows

The debugger provides a number of default windows that allow you to treat the entire screen as a single window (FS) or divide the screen into halves (H1,H2), thirds (T1,T2,T3), quarters (Q1,Q2,Q3,Q4), or sixths (S1,S2,S3,S4,S5,S6). The full screen (FS) covers lines 1-24; H1 covers lines 1-12 and H2 covers lines 13-24. In addition, each of these screens can also be divided vertically into a right and left half. For example, the top right side of the screen (RH1) is where INST and REG are displayed and the top left side of the screen (LH1) is where SCR is displayed.

Use the SHOW WINDOW command to display the names and dimensions of all windows currently defined. Use the CANCEL WINDOW command to delete one or more windows.

5.3.3 Manipulating Displays

You can manipulate screen displays in several ways, including showing them on the screen, scrolling forward or backward through them, and removing them from the screen.

5.3.3.1 Showing Displays

You can show a REG or INST display on the right side of the screen with the DISPLAY command. For example, in response to the DISPLAY REG command, the debugger places the REG display in window RH1. If the DISPLAY INST command is issued, the REG display is replaced with the INST display in RH1.

```
DBG> DISPLAY REG
DBG>
```

```
 --SRC: module INCOME---scroll-source-------------------------------------
   219:          ! Declare subprograms in|R0:00007C00  R10:7FFEDD4  @SP:00000000
   220:          2       CLI$PRESENT,    |R1:00000000  R11:00006904  +4:08000020
   221:          2       LIB$GET_LUN     |R2:00000000  AP :7FF557CC  +8:7FF55FCC
   222:                                  |R3:7FF55F94  FP :7FF55740 +12:7FF55758
   223:          ! Get logical unit numbe|R4:00000000  SP :7FF55740 +16:00009B5E
-> 224:          STATUS = LIB$GET_LUN (ST|R5:00000000  PC :00007C09 +20:7FFE33DC
   225:          IF (.NOT. STATUS) CALL L|R6:7FF55449  @AP:00000000 +24:00000000
   226:                                  |R7:8001E4DD    +4:800212C0 +28:20000000
   227:          ! Get name of file conta|R8:7FFED052    +8:20000000 +32:7FF557CC
   228:          CALL CLI$GET_VALUE ('STA|R9:7FFED25A  +12:00000000 +36:7FF55FF0
   229:          2                   STAT|N:0    Z:0     V:0    C:0 +40:0000924B
 --OUT---out------------------------------------------------------------
display SRC at H1, size = 64
    kind = SOURCE (EXAMINE/SOURCE .%SOURCE_SCOPE\%PC)
display INST at H1,size = 64, removed
    kind = INSTRUCTION (EXAMINE/INSTRUCTION .0\%PC)
display REG at RH1, size = 64, removed, kind = REGISTER
display OUT at S45, size = 100, kind = OUTPUT
display PROMPT at S6, size = 64, kind = PROGRAM
-PROMPT -error-program-prompt-----------------------------------------------
DBG>

----------------------------------------------------------------------------
```

5.3.3.2 Removing Displays

If you remove a display from the screen and that display is overlying another display, the hidden display appears in place of the removed display. If you remove a display from the screen and that display is not overlying any other displays, the contents of the display remain on the screen until overwritten by subsequent debugger and/or program output. The following commands remove a display from the screen.

- CANCEL DISPLAY—Removes the display from the screen and deletes it from memory. For example, to delete display OUT, specify the following:

  ```
  DBG> CANCEL DISPLAY OUT
  ```

  If you delete the OUT display, the debugger no longer sends output to a special display; instead, command output follows the command line.

- DISPLAY/REMOVE—Removes the display from the screen and saves it (as is) in memory. For example, to save display OUT, specify the following:

  ```
  DBG> DISPLAY/REMOVE OUT
  ```

  To reactivate the OUT display, use the DISPLAY command to return the display to the screen (see Section 5.3.3.1) and the SELECT/OUTPUT command to select the OUT display for debugger output (see the description of NORMAL displays in Section 5.3.4).

- DISPLAY/HIDE—Determines whether a specified display is overlying another display. If so, the debugger uncovers the hidden display hiding the specified display; otherwise, the display remains unchanged. The debugger saves and continues to update the original display even if it is hidden as a result of the DISPLAY/HIDE command. You can return the original display to the screen with the DISPLAY command. For example, to force any display hidden by the OUT display to appear on the screen, specify the following:

  ```
  DBG> DISPLAY/HIDE OUT
  ```

You can use the DISPLAY/CLEAR command to erase the contents of a display without removing the display from the screen.

5.3.3.3 Scrolling Displays

The SCROLL command allows you to show different parts of a display on the screen. The SELECT/SCROLL command determines which display is affected by the SCROLL command. For example, to make the SRC display the default scrolling display, specify the following:

```
DBG> SELECT/SCROLL SRC
```

Display SRC remains the default parameter of the SCROLL command until you specify another SELECT/SCROLL command. To display the previous five lines of source code, specify the following:

```
DBG> SCROLL/UP:5
```

Typically, you scroll a display using the keypad keys.

- Up—Key 8 scrolls towards the beginning of the display by entering the SCROLL/UP command. To scroll to the top of the display, press PF1 followed by key 8.
- Down—Key 2 scrolls towards the end of the display by entering the SCROLL/DOWN command. To scroll to the bottom of the display, press PF1 followed by key 2.
- Left—Key 4 scrolls towards the left of the display by entering the SCROLL/LEFT command.
- Right—Key 6 scrolls toward the right of the display by entering the SCROLL/RIGHT command.

Keypad key 5 refreshes the current source display (SRC, by default) causing the next source line that is to be executed to appear in the middle of the display (the source line is marked with an arrow at the left of the source display).

5.3.3.4 Using Pseudodisplay Names

The debugger keeps a chronological list of the displays that you reference with DISPLAY commands. You can refer to displays by their relative positions in this list; that is, you can refer to the current (most recently referenced) display or the next display. (You can restrict the list to a single type of display as well.)

For example, %NEXTDISP is a pseudodisplay name that refers to the next display in the list. That is, if you create displays A, B, and C in that order and you are currently displaying B, the name %NEXTDISP refers to display C. If you display A while B is current, A becomes current and B moves to the end of the list, making the order of the list A, C, B. The default definition for keypad key 9 is DISPLAY %NEXTDISP, which allows you to use the 9 key to circle through your displays (assuming that you have referenced each display with the DISPLAY command).

You can use the following pseudodisplay names to reference displays in debugger commands.

%CURDISP	The current (most recently referenced) display
%CURSCROLL	The current scrolling display
%NEXTDISP	The next display in the list
%NEXTINST	The next instruction display
%NEXTOUTPUT	The next output display
%NEXTSCROLL	The next scrolling display
%NEXTSOURCE	The next source display

5.3.4 Creating Displays

In addition to the default displays, you can define other source, output, and register displays. To create a display, use the SET DISPLAY command.

```
SET DISPLAY display [AT window] [type]
```

You must specify a display name; optionally, you can specify the window into which the display is mapped and the type of display to create. The following display types are available.

- DO (command-list)—Display contains the results of the debugger commands specified in the command list. The command list is executed each time the debugger regains control. If you specify more than one command, separate the commands using semicolons.

- INSTRUCTION—Display contains the assembly language instructions, but only if the display is selected for instruction display with the SELECT /INSTRUCTION command.
- INSTRUCTION (command-list)—Display contains the results of the debugger commands specified in the command list. The command list, which should consist of a single EXAMINE/INSTRUCTION command, is executed each time the debugger regains control. The SRC display is type INSTRUCTION with a command list of EXAMINE/INSTRUCTION .0\%PC (examine the instruction line in the current module at the location in the program counter). By default, the INST display is selected for instruction display when the language is MACRO.
- OUTPUT—Display contains all debugger output, but only if the display is selected for output with the SELECT/OUTPUT command. By default, the OUT display is selected for output.
- REGISTER—Display contains the VAX registers and their contents; the display is updated each time the debugger regains control. The REG display is type REGISTER.
- SOURCE—Display contains the program source statements, but only if the display is selected for source display with the SELECT/SOURCE command.
- SOURCE (command-list)—Display contains the results of the debugger commands specified in the command list. The command list, which should consist of a single TYPE or EXAMINE/SOURCE command, is executed each time the debugger regains control. The SRC display is type SOURCE with a command list of EXAMINE/SOURCE .0\%PC (examine the source line in the current module at the location in the program counter). By default, the SRC display is selected for source display for all language except MACRO.

5.4 Controlling Program Execution

Debugger commands provide several means of controlling your program's execution. If your program contains more than one program unit, you may need to use the SET MODULE and/or SET SCOPE commands in order to reference symbols in all modules. (See Sections 5.5.1 and 5.5.3.2 for information about the SET MODULE and SET SCOPE commands.) For a complete description of each debugger command, see the *VAX/VMS Debugger Reference Manual.*

5.4.1 Starting Program Execution

The command with which you start program execution determines when the debugger regains control. The GO, STEP, and CALL commands execute varying numbers of source lines before returning control to the debugger. The SHOW CALLS command displays the current hierarchy of routine calls.

5.4.1.1 The GO Command

The GO command starts execution at the current line and continues to the conclusion of the program (as in the following example), an error, or the next breakpoint or watchpoint (see Section 5.4.2).

```
DBG> GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

An optional parameter of the GO command allows you to specify an address at which to start program execution. This feature allows you to reexecute your program without exiting from the debugger; however, unless you reinitialize modified data, your results are likely to be incorrect.

5.4.1.2 STEP Command

By default, the STEP command executes one source statement. To execute more than one statement, specify the number of statements to be executed as a parameter of the STEP command. The following command executes the next three source statements.

```
DBG> STEP 3
stepped to INCOME\%LINE 228
   228:   CALL CLI$GET_VALUE ('STATS_FILE',
```

If you are debugging in screen mode, the information displayed by the STEP command is redundant since the SRC display shows the source code and your current position. Use the SET STEP SILENT command to prevent the STEP command from displaying any text.

If the STEP command encounters a subprogram invocation, the following STEP characteristics determine whether the subprogram executes as a single step.

- OVER—Causes the debugger to execute subprograms as a single STEP command.
- INTO—Causes the debugger to step through subprograms line by line.

- NOSYSTEM and SYSTEM—NOSYSTEM causes the debugger to execute system subprograms as a single STEP command regardless of the INTO and OVER characteristics. SYSTEM and INTO together cause the debugger to step through system subprograms line by line.

To display the current default characteristics of the STEP command, enter the SHOW STEP command. The following SHOW STEP command displays the default STEP parameters.

```
DBG> SHOW STEP
STEP TYPE:  NOSYSTEM, SOURCE, OVER ROUTINE CALLS, BY LINE
```

By default, the debugger executes all subprograms as a single step. To set the STEP characteristics so that the debugger steps through user subprograms, but not system subprograms, specify the following:

```
DBG> SET STEP NOSYSTEM, SOURCE, INTO, LINE
```

The LINE characteristic in the previous example (same as BY LINE in the SHOW STEP display) indicates that a STEP command will execute source line by source line. Alternatively, you can use the following keywords to specify that a STEP command should execute all lines up to an exception or a particular type of machine code instruction (the BRANCH, CALL, and INSTRUCTION keywords are useful only if you are familiar with machine code).

- BRANCH—Specifies the next machine code branch instruction
- CALL—Specifies the next machine code call instruction
- EXCEPTION—Specifies the next exception (error)
- INSTRUCTION—Specifies the next machine code instruction
- LINE—Specifies the next source code statement
- RETURN—Specifies the end of the currently executing subprogram

To use one of the previously listed keywords to modify a single STEP command, name the keyword as a qualifier of the STEP command (for example, STEP/INSTRUCTION). To use one of the previously listed keywords as the default for the STEP command, use the keyword as a parameter of the SET STEP command (for example, SET STEP INSTRUCTION).

5.4.1.3 CALL Command

The CALL command invokes a subprogram (passing it specified arguments), executes the subprogram, and displays the function value returned (0 for a subroutine). Typically, you use the CALL command to invoke a subprogram that you have written to display data structures or other information required for debugging.

When passing arguments to a subprogram invoked by the CALL command, use the following keywords to specify the passing mechanism.

- %ADDR (default)—Pass by address; typically, you use this mechanism when passing the name of a routine as an argument
- %DESCR—Pass by descriptor
- %REF—Pass by reference
- %VAL—Pass by value

For example, the subprogram INC_DUMP is a subroutine that requires two arguments, both passed by reference. To invoke INC_DUMP, specify the following:

```
DBG> CALL INC_DUMP (%REF PERSONS_HOUSE, %REF ADULTS_HOUSE)
value returned is 0
```

5.4.1.4 SHOW CALLS Command

To display information about the sequence of subprogram calls, or the number of call frames on the stack, use the SHOW CALLS command. For each call frame (beginning with the most recent call), the debugger displays one line of information, including the name of the routine, the name of the module containing the routine, and the line number of the call. For example:

```
DBG> SHOW CALLS
 module name   routine name        line    rel PC    abs PC
*CALC_SUMS     CALC_SUMS                  00000002  00008D06
*REPORT        REPORT                68   0000009A  00008C52
*INCOME        INCOME               253   00000276  00008676
```

The value of the program counter (PC) in the calling routine at the time that control passed to the called routine is also displayed. The PC is expressed both as a virtual address relative to the virtual address of the routine's name and as an absolute address.

5.4.2 Suspending (or Tracing) Program Execution

You can suspend program execution at specified locations in your program by setting breakpoints and watchpoints with the SET BREAK and SET WATCH commands. In addition, you can follow program execution without suspending execution by setting tracepoints.

An optional WHEN clause allows you to activate a breakpoint, tracepoint, or watchpoint conditionally. An optional DO clause allows you to specify one or more debugger commands to be executed when a breakpoint, tracepoint, or watchpoint is activated. SHOW and CANCEL commands display and cancel the breakpoints, tracepoints, and watchpoints that you have set.

Note

You cannot set a breakpoint, tracepoint, and watchpoint at the same location: the most recently issued command overrides any other breakpoint, tracepoint, or watchpoint at that location.

5.4.2.1 Breakpoints and Tracepoints

You can set a breakpoint at a particular program location, on an exception, or on a particular type of instruction. When your program encounters a breakpoint, the debugger suspends program execution, displays the address of the breakpoint and the source line at that address, executes the DO command sequence (if specified), and (unless the DO command sequence causes an alternative action) prompts for a command.

A tracepoint is exactly like a breakpoint, except that instead of prompting for a command, the debugger executes an implicit GO command to continue program execution.

To set a breakpoint or tracepoint at a particular program location, specify that location as the parameter of the SET BREAK or SET TRACE command. The following example sets a breakpoint that causes the debugger to suspend execution just before line 224 in module INCOME.

```
DBG> SET BREAK INCOME\%LINE 224
DBG> GO
DBG> STEP
break at INCOME\%LINE 224
    224:            STATUS = LIB$GET_LUN(STATS_LUN
```

A breakpoint usually suspends execution at the first byte of the specified location so that the instruction beginning at that location does not execute. However, if you set a breakpoint at a routine, the breakpoint is actually set at the memory address two bytes greater than the address of the routine name itself (the entry point), thereby causing the routine to be called before the debugger takes control and issues the message “routine break at routine NAME”.

To set a breakpoint or tracepoint on an exception or type of instruction, use the following qualifiers of the SET BREAK and SET TRACE commands (note that these qualifiers are the same as those used on the STEP and SET STEP commands).

- /BRANCH—Specifies the next machine code branch instruction.
- /CALL—Specifies the next machine code call instruction.
- /EXCEPTION—Specifies the next exception (error). The debugger reports the exception and the line at which it occurred, after which you can execute or inhibit a user-declared condition handler. You cannot use the STEP command to step into a condition handler, but you can set a breakpoint or tracepoint within the handler. (The SET BREAK /EXCEPTION command is the same as the SET EXCEPTION BREAK command.)
- /INSTRUCTION—Specifies the next machine code instruction.
- /LINE—Specifies the next source code statement.
- /RETURN—Specifies the end of the currently executing subprogram.

You can use tracepoints (or breakpoints) to indicate both the order and frequency with which subprograms are called by setting a tracepoint at every subprogram. (If you use the /CALL qualifier, a tracepoint is placed on every machine code call instruction, which is typically many more tracepoints than you want.) For example, to trace subprogram calls for INCOME (assuming that INCOME invokes the subprograms GET_STAT, FIX_STAT, and REPORT), specify the following:

```
DBG> SET TRACE GET_STAT
DBG> SET TRACE FIX_STAT
DBG> SET TRACE REPORT
DBG> GO
```

To activate a tracepoint or breakpoint exactly once, specify the /TEMPORARY qualifier. To activate a tracepoint or breakpoint after a certain number of iterations, specify the /AFTER qualifier. For example, to activate a tracepoint on the third execution of line 35 in module CALC_SUMS, specify the following:

```
DBG> SET TRACE/AFTER:3 CALC_SUMS\%LINE 35
DBG> GO
trace at CALC_SUMS\%LINE 35
   35            PERSONS = PERSONS + PERSONS_HOUSE(I)
trace at CALC_SUMS\%LINE 35
   35            PERSONS = PERSONS + PERSONS_HOUSE(I)
                   .
                   .
                   .
```

The tracepoint is activated at each successive execution of that line (unless you specify /TEMPORARY).

To have a breakpoint or tracepoint execute a list of commands when it is activated, use a DO command sequence. The following example sets a breakpoint on line 242 and, if the value of TOTAL_HOUSES is more than 100, displays the value of TOTAL_HOUSES; if not, execution resumes at line 243.

```
DBG> SET BREAK INCOME\%LINE 242 DO (IF TOTAL_HOUSES .GT. 100 -)
_ THEN (EXAMINE TOTAL_HOUSES) ELSE (GO)
```

The optional WHEN clause allows you to conditionally execute a breakpoint or tracepoint. Each time the debugger encounters the breakpoint or tracepoint, it evaluates the expression in the WHEN clause: if the expression is true, the breakpoint or tracepoint is activated; if it is false, the breakpoint or tracepoint is ignored. For example, in the following command, the DO command sequence executes only if the expression TOTAL_HOUSES .GT. 100 is true.

```
DBG> SET TRACE INCOME\%LINE 242 WHEN(TOTAL_HOUSES .GT. 100) -
_ DO (EXAMINE PERSONS)
```

5.4.2.2 Watchpoints

Specify a watchpoint to display a particular program location each time the contents of that location are modified. When your program modifies the specified location, the debugger suspends program execution; displays the address of the location, the old and new values of the location, and the source line that modified the location; executes the DO command sequence (if specified); and (unless the DO command sequence causes an alternative action) prompts for a command. To set a watchpoint on a location, specify that location as the parameter of the SET WATCH command. The following example sets a watchpoint on the location TOTAL_HOUSES in the module INCOME and starts execution.

```
DBG> SET WATCH INCOME\TOTAL_HOUSES
DBG> GO
watch of INCOME\TOTAL_HOUSES at 50340
        old value = 0
        new value = 65
break at 50345
DBG>
```

The SET WATCH command uses the /AFTER and /TEMPORARY qualifiers in the same way the SET BREAK and SET TRACE commands do. Likewise, you can execute a list of commands when a watchpoint is activated by using a DO command sequence, or you can conditionally activate a watchpoint by using a WHEN clause. See Section 5.4.2.1 for examples of the /AFTER and /TEMPORARY qualifiers, as well as the DO and WHEN clauses.

5.4.3 Displaying Source Lines

Debugger commands allow you to display source lines both when they execute and independent of their execution. The STEP/SOURCE and SET MODE SCREEN commands display source lines as they execute; the TYPE and SEARCH commands display source lines independently. All commands that display source lines require the following:

1. The source file must have been compiled with the /DEBUG and /NOOPTIMIZE qualifiers.
2. The source file must reside in the same location in which it was compiled, or you must use the SET SOURCE command to establish the file's new location. For example, if the source file has been moved to the subdirectory [INCOME.SOURCE], specify the following:

   ```
   DBG> SET SOURCE [INCOME.SOURCE]
   ```
3. You must specify the appropriate scope (or path-name) when necessary.

The STEP/SOURCE command displays the currently executing source lines; see Section 5.4.1.2 for information about the STEP command. The SET MODE SCREEN command generates the SRC display by default; SRC shows the currently executing source line, the lines preceding it, and the lines following it; see Section 5.3 for information on debugger screen displays. The TYPE and SEARCH commands display source lines independent of their execution.

- TYPE—Displays a specified range of source lines. For example, to display lines 22 through 27 of module INCOME, type

  ```
  DBG> TYPE INCOME\22:27
  module INCOME
      22:         2  ADULTS_HOUSE (2048),
      23:         2  INCOME_HOUSE (2048)
      24:
      25:         ! Declare variables and values
      26:         INTEGER STATUS,
      27:         2       IOSTAT
  ```

  In SCREEN mode, the source lines appear in the source display (SRC, by default); in NOSCREEN mode, the source lines appear with other debugger output.

- SEARCH—Displays the source lines containing the specified string. For example, to display all lines containing the string STATS_FILE in the module named INCOME, specify the following:

```
DBG> SEARCH/ALL INCOME STATS_FILE
module    INCOME
    213:      CHARACTER*255 STATS_FILE
    228:      CALL CLI$GET_VALUE (STATS_FILE',
    229:      2                   STATS_FILE,
    238:      2    FILE=STATS_FILE (1:SF_SIZE,
```

The SEARCH display appears with other debugger output.

5.4.4 Using Logical Control Structures

You can further control the flow of your program's execution by using the following debugger commands for logical control.

- IF THEN ELSE—A conditional construction that executes a THEN clause of one or more debugger commands if a specified logical expression is true. If the expression is false, the command either terminates or executes an optional ELSE clause of one or more debugger commands. The format of an IF construction is as follows:

  ```
  IF expression THEN (command[;...]) [ELSE(command[;...])]
  ```

  In the following example, if the expression in the IF clause is false, the ELSE clause executes.

  ```
  DBG> IF I .GT. 100 THEN (EXAMINE PERSONS) ELSE (GO)
  ```

- WHILE—An iterative construction that executes a DO command sequence while a specified logical expression is true; if the expression is false, the command terminates. The format of a WHILE construction is as follows:

  ```
  WHILE expression DO(command[;...])
  ```

  The following example causes the debugger to step line by line while TOTAL_HOUSE is less than 5.

  ```
  DBG> WHILE TOTAL_HOUSES .LT. 5 DO(STEP/LINE)
  ```

- FOR—An iterative construction that executes a DO command sequence through a range of values. The format of a FOR construction is as follows:

  ```
  FOR control = init TO term [BY inc] DO(command[;...])
  ```

  The control variable is initialized to the value of **init** and compared to **term**. If **control** is less than **term**, the commands in the DO command sequence execute and **control** is incremented by one (or the value of **inc**). If **control** is greater than **term**, the command terminates. For example, the following FOR command displays the value of three variables five times.

```
    DBG> FOR I = 1 TO 5 DO(EXAMINE PERSONS,ADULTS,INCOME)
```

Logical control structures are especially useful in command procedures and DO command sequences (see Section 5.7 for information about using debugger command procedures). The following command sets a break on line 36 of module CALC_SUMS; evaluates an expression; and, if true, executes a command procedure, or, if false, continues execution at line 36.

```
DBG> SET BREAK CALC_SUMS\%LINE 36 DO(IF I .GT. 100 -)
_ THEN (EXAMINE TOTAL_HOUSES) ELSE (GO)
```

5.5 Symbolic Debugging

During a debugging session you can reference the following items:

- Locally defined program variables, such as variables
- Globally defined program variables, such as routine names
- Virtual memory locations, such as an address returned by a system-defined routine
- Debugger symbols for VAX registers:

%R0—%R11	General registers 0 through 11
%AP	Argument pointer
%FP	Frame pointer
%SP	Stack pointer
%PC	Program counter
%PSL	Processor status longword

- User-defined debugger symbols as described in Section 5.5.4.

5.5.1 Maintaining Symbol Information

The debugger maintains information about the local and global symbols in your program, depending upon the DCL commands used to compile and link the program.

- Local symbols—To maintain local symbol information (symbolic names of variables) in a FORTRAN program, specify the /DEBUG qualifier with both the FORTRAN and LINK commands.

```
FORTRAN/DEBUG/NOOPTIMIZE
LINK/DEBUG
```

- Global symbols—To maintain global symbol information (symbolic names of routines in a FORTRAN program, procedure entry points, and global data names), specify the /DEBUG and /NOOPTIMIZE qualifiers with the LINK command.

  ```
  $ LINK/DEBUG
  ```

By default, only symbols declared in the main program unit are available to the debugger at run-time. To make the symbols of any other program unit available, use the SET MODULE command. For example, to make the symbols in the module GET_STATS available, specify the following:

```
DBG> SET MODULE GET_STATS
```

If your program is not too large, it is useful to make the symbols of all modules available at the outset of the debugging session. (Performance will suffer noticeably if your program is too large for all of its modules to be set.) To set all modules, specify the /ALL qualifier with the SET MODULE command.

```
DBG> SET MODULE/ALL
```

Depending upon the size of your program, most or all of its modules' symbols will be set. To display which modules' symbols are available, use the debugger command SHOW MODULE. The following example demonstrates the effect of the SET MODULE/ALL command. At the outset of a debugging session on a FORTRAN program named INCOME, only the symbols of the first linked module (the main program unit named INCOME) are available.

```
DBG> SHOW MODULE
module name              symbols      language    size

INCOME                     yes         FORTRAN     948
CONVERT_FIXES              no          FORTRAN     424
FIX_STATS                  no          FORTRAN     372
GET_STATS                  no          FORTRAN      60
REPORT                     no          FORTRAN     660
CALC_SUMS                  no          FORTRAN     664
GET_1_STAT                 no          FORTRAN     264
PUT_PART1                  no          FORTRAN     496
PUT_PART2                  no          FORTRAN     704
PUT_RPTHEAD                no          FORTRAN     200
LEFT_JUSTIFY               no          FORTRAN     284

total FORTRAN modules: 11    remaining size: 55856
```

Entering the SET MODULE/ALL command makes all the modules' symbols available.

```
DBG> SET MODULE/ALL
DBG> SHOW MODULE
module name               symbols    language     size
INCOME                      yes         FORTRAN      948
CONVERT_FIXES               yes         FORTRAN      424
FIX_STATS                   yes         FORTRAN      372
GET_STATS                   yes         FORTRAN      660
REPORT                      yes         FORTRAN      660
CALC_SUMS                   yes         FORTRAN      664
GET_1_STAT                  yes         FORTRAN      264
PUT_PART1                   yes         FORTRAN      496
PUT_PART2                   yes         FORTRAN      704
PUT_RPTHEAD                 yes         FORTRAN      200
LEFT_JUSTIFY                yes         FORTRAN      284
total FORTRAN modules: 11    remaining size: 50964
```

If your program is not too large, including the SET MODULE/ALL command in an initialization file (see Section 5.7.3) is useful for two reasons. First, it makes the symbol records of all modules automatically available. Second, if you use the SET MODULE/ALL command before your own program gains control, program data and debugger data will be stored separately, eliminating the possibility of interspersing program memory with debugger memory. (Interspersing program and debugger memory is not a problem unless the program is dependent on the exact location of the data in memory.)

Note that the SHOW MODULE command displays the remaining size of the debugger's storage area. You can increase this storage area by using the ALLOCATE command or the /ALLOCATE qualifier of the SET MODULE command. However, both the /ALLOCATE qualifier and the ALLOCATE command may cause program and debugger memory to be interspersed.

In addition to specifying the DCL /DEBUG qualifier at compile and link time and the debugger SET MODULE command at debug time, you may also need to use the debugger SET SCOPE command to reference symbols during the debugging session; see Section 5.5.3.2.

5.5.2 Referencing Symbols

You can reference a symbol by specifying the symbol name as a parameter in debugger commands. The following command displays the value contained in the variable VM_SIZE.

```
DBG> EXAMINE VM_SIZE
V$MAIN\VM_SIZE: 24580
```

Generally, a symbol name is equated to an address; therefore, specifying the symbol name is the same as specifying the address. The following sequence of commands displays the address associated with the symbol VM_SIZE and then displays the value at that address.

```
DBG> EVALUATE/ADDRESS VM_SIZE
25608
DBG> EXAMINE 25608
V$MAIN\VM_SIZE: 24580
```

The debugger also provides the SYMBOLIZE command to allow you to determine the symbol, if any, associated with a particular location in memory.

```
DBG> SYMBOLIZE 25608
address 00006408:
    V$MAIN\VM_SIZE
```

When examining memory locations, you can also use the following shorthand notations:

. (period)	Current location (the location most recently referenced by an EXAMINE or DEPOSIT command)
^ (circumflex)	Previous location (the location at the next lower address from the current location)
RETURN	Next location (the location at the next higher address from the current location;) invalid for the DEPOSIT command

5.5.3 Resolving Symbol References

Given that the required symbols are available to the debugger, you may need to specify the program region, or scope, in which a particular symbol is to be interpreted. Scope is dynamic, changing by default to the module that is currently executing. To display the current default scope (represented both by number and module name), enter the SHOW SCOPE command.

```
DBG> SHOW SCOPE
scope:  0   [ = INCOME]
```

A numeric scope of 0 indicates that the scope is the currently executing program unit; a numeric scope of 1 indicates that the scope is the program unit that invoked the currently executing program unit; and so on.

When the debugger encounters a reference to a symbol, it attempts to resolve that symbol with the following steps:

1. If the symbol name is unique within the program, the debugger can reference its definition.
2. If the symbol is not unique within the program, but is used within the current scope, then the debugger uses the definition for the symbol as defined by the current scope.
3. If the symbol is not defined within the program, the debugger issues a message indicating that the symbol is "not in the symbol table". In this case, you might have misspelled the symbol, forgotten to use the SET MODULE command to include the symbols of a particular module, forgotten the /DEBUG qualifier when you compiled or linked the program, or linked the FORTRAN program with the /OPTIMIZE instead of the /NOOPTIMIZE qualifier.

   If the symbol is not unique within the program and is not used within the current scope, the debugger issues a message indicating that the symbol "is not unique." In this case, you must specify a path-name or use the SET SCOPE command, as shown in the following sections, to resolve the ambiguity for the debugger. (If necessary, use the SHOW SYMBOL command to list the modules that define the symbol.)

5.5.3.1 Path-Name Prefix

You can make a symbol unique by specifying a string of symbolic names connected by backslashes that fully identify the symbol. The string, or path-name, can include the module, routine, block, labeled section, and/or line that contains it. The path-name can be incomplete, so long as it makes the symbol unique. Usually one path-name prefix is sufficient to make the symbol unique. For example, to specify the variable LINE_NO in the module GET_STATS (rather than the variable LINE_NO in the module GET_1_STAT), specify the following:

```
DBG> EXAMINE GET_STATS\LINE_NO
```

Examples of other path-names are:

```
DBG> EXAMINE %LINE 121\LINE_NO
DBG> EXAMINE GET_STATS\%LINE 121\LINE_NO
DBG> EXAMINE O\LINE_NO
```

Modules in path-names can be specified numerically (as in the preceding example), where the currently executing module is 0, the module that calls the currently executing module is 1, the module that calls the module that calls that one is 2, and so on.

5.5.3.2 SET SCOPE Command

You can use the SET SCOPE command to specify one or more program regions to be used by default in the interpretation of symbols. For example, to make the module GET_STATS the default scope, specify the following:

```
DBG> SET SCOPE GET_STATS
```

Subsequently, the path-name GET_STATS\ is the default prefix of references to symbols without path-names.

```
DBG> EXAMINE LINE_NO
GET_STATS\LINE_NO:          0
```

You can also use a list of modules or path-names as a parameter of the SET SCOPE command to set the order in which the debugger searches for referenced symbols. For example, the following command makes the debugger search first the module GET_STATS and then the module GET_1_STAT for whatever symbol is being referenced.

```
DBG> SET SCOPE GETSTATS,GET_1_STAT
DBG> SHOW SCOPE
scope: GET_STATS, GET_1_STAT
```

5.5.4 Defining Symbols

You can assign a symbolic name to a program location, value, or character string with the debugger command DEFINE. You might, for instance, define a symbol to represent a frequently referenced location that is hard to remember. The following example assigns the symbolic name TOT to the integer variable TOTAL_HOUSES in the module INCOME and then references the variable by its assigned name.

```
DBG> DEFINE TOT = INCOME\TOTAL_HOUSES
DBG> EXAMINE TOT
INCOME\TOTAL_HOUSES:  57
```

The symbol definition lasts for the duration of the debugging session or until you cancel it with the UNDEFINE command.

5.5.5 Displaying Symbol Information

To display information about the symbols in your program, use the SHOW SYMBOL command. For example, to display the address and type of all symbols named INCOME in a FORTRAN program, enter

```
DGB> SHOW SYMBOL/ADDRESS/TYPE INCOME
data CALC_SUMS\INCOME
    address:  +290690
    atomic type, F_floating, size: 4 bytes
data GET_STATS\INCOME
    address:  +27556
    atomic type, longword integer, size: 4 bytes
routine INCOME
    address:  31744, size 763 bytes
routine INCOME (global)
    address:  31744
module INCOME, language FORTRAN
```

To display information about symbols you have defined during the debugging session, use the SHOW SYMBOL/DEFINED command.

```
DBG> SHOW SYMBOL/DEFINED TOT
    bound to:  INCOME\TOTAL_HOUSES
    was defined /address
```

By default, the SHOW SYMBOL command returns information about global symbols and those symbols in modules that have been set (either by default or with the SET MODULE command). The IN clause allows you to restrict the SHOW SYMBOL command to one or more modules. You can use the asterisk wildcard character with the SHOW SYMBOL command to match any number of characters in the symbol's name. The following command displays information about all symbols within the scope of the module CALC_SUMS (any specified scope must be in a module that is set in order for the SHOW SYMBOL command to work properly).

```
DBG> SHOW SYMBOL * IN CALC_SUMS
data CALC_SUMS\TOTAL_HOUSES
data CALC_SUMS\PERSONS_HOUSE
data CALC_SUMS\ADULTS_HOUSE
data CALC_SUMS\INCOME_HOUSE
data CALC_SUMS\AVG_PERSONS_HOUSE
data CALC_SUMS\AVG_ADULTS_HOUSE
data CALC_SUMS\AVG_INCOME_HOUSE
data CALC_SUMS\AVG_INCOME_PERSON
data CALC_SUMS\MED_INCOME_PERSON
data CALC_SUMS\PERSONS
data CALC_SUMS\ADULTS
data CALC_SUMS\INCOME
data CALC_SUMS\MEDIAN
data CALC_SUMS\I
data CALC_SUMS\J
```

To display the address specification of all symbols beginning with the characters LIB$, specify the following:

```
DBG> SHOW SYMBOL/ADDRESS LIB$*
routine LIB$DATE_TIME (global)
    address:    00009994
```

5.6 Manipulating Data

The debugger allows you to examine and manipulate data as your program executes. (If the locations referenced by the following commands are not in your default scope, you must set the module and specify a scope or path-name, as described in Section 5.5.3.)

When using commands that manipulate data (such as EXAMINE and DEPOSIT), be aware of the difference between an address expression and a language expression.

- Address expression—An address expression specifies a program location. If the location is that of a symbol defined by the source program, it has a language-dependent data type associated with it; otherwise, no data type is associated with it. An address expression may consist of a single operand or multiple operands combined with the debugger operators; it is evaluated in the following order:

  1. Parenthesized parts of the expression
  2. Operators by rank low to high (see the following table)
  3. Operators of the same rank from left to right

  The result of an address expression is a 32-bit longword integer that represents a program location.

- Language expression—A language expression specifies a value; the value is associated with a data type. A language expression may consist of a single operand or multiple operands combined with operators; the expression is evaluated according to the rules of precedence for the source language. The result of a language expression must be a value that is valid for the current source language.

For example, assume that you have a symbol named NUMBER that has a value of 3 and is located at address 1600. The EXAMINE command, which expects an address expression, interprets (NUMBER + 1) as 1601. The EVALUATE command, which expects a language expression, interprets (NUMBER + 1) as 4.

Operator	Rank	Description
. or @	1	Unary operators specifying the contents of the operand
+ or –	1	Unary operators specifying the positive or negative value of the operand
* or /	2	Binary operators specifying the multiplication or division of the operands
+ or –	3	Binary operators specifying the addition or subtraction of the operands

5.6.1 Displaying Values

The EXAMINE command displays the contents of a specified program location. For example, to display the contents of the variable ADULTS in the module CALC_SUMS, set the module to CALC_SUMS and specify the following:

```
DBG> EXAMINE TOTAL_HOUSES
CALC_SUMS\TOTAL_HOUSES:  16
```

To display array elements, specify the elements individually (1:1, 2:2, etc.), in a range (1:10), or with a wildcard (*). To display an entire array, specify the array name without a subscript. The following command displays the first ten elements of the one-dimensional array PERSONS_HOUSE.

```
DBG> EXAMINE PERSONS_HOUSE(1:10)
CALC_SUMS\PERSONS_HOUSE
    (1):        6.000000
    (2):        3.000000
    (3):        10.00000
    (4):        6.000000
    (5):        1.000000
    (6):        4.000000
    (7):        3.000000
    (8):        2.000000
    (9):        3.000000
    (10):       4.000000
```

You can also use the EXAMINE command to display a record field or an entire record. If you choose to display the entire record, fields are displayed in order (as described by the STRUCTURE block). Substructure and array fields are fully displayed, as are all maps in all unions. The following example displays the record CARRIAGE, and then displays the CUSTOMER field of that same record. (For clarity, the record definition is shown before the debugger session.)

```
STRUCTURE /STOCK/
 INTEGER*4 STOCK_NUMBER
 STRUCTURE /PRODUCT_CODE/ CODE
  CHARACTER*12 ITEM
  INTEGER*4    PART_NUMBER
 END STRUCTURE
 UNION
  MAP
   CHARACTER*30 CUSTOMER
  END MAP
  MAP
   CHARACTER*30 DEPARTMENT
  END MAP
 END UNION
END STRUCTURE
RECORD /STOCK/ CARRIAGE

DBG> EXAMINE CARRIAGE
INVENTORY\CARRIAGE
     STOCK_NUMBER:      48796
     CODE
          ITEM:   "TRANSPORT   "
          PART_NUMBER:   12
     CUSTOMER:    "BENNINGTON GARAGE             "
     DEPARTMENT:  "..............................."
DBG> EXAMINE CARRIAGE.CUSTOMER
INVENTORY\CARRIAGE.CUSTOMER:    "BENNINGTON GARAGE             "
```

5.6.2 Calculating Values

The EVALUATE command displays the value of a specified language expression. For example, to add the value in PERSONS and the value in PERSONS_HOUSE(7), where both symbols are defined in the module CALC_SUMS, set module and scope to CALC_SUMS and specify the following:

```
DBG> EVALUATE PERSONS + PERSONS_HOUSE(7)
3.000000
```

Using the EVALUATE command you can also perform arithmetic calculations that may or may not be related to your program, in effect using the debugger as a calculator.

5.6.3 Assigning Values

The DEPOSIT command assigns a language expression to a program location. For example, to assign the value 5 to the array element PERSONS_HOUSE(7) in module CALC_SUMS, set module and scope to CALC_SUMS and specify the following:

```
DBG> DEPOSIT PERSONS_HOUSE(7) = 5
```

Subsequent examination of the variable displays its assigned value.

```
DBG> EXAMINE PERSONS_HOUSE(7)
CALC_SUMS\PERSONS_HOUSE(7):   5
```

5.6.4 Specifying Data Type

Typically, when you examine a program location, you want to use the data type that your program has associated with that location. For example, if you have defined STATUS as a variable of data type INTEGER, when you examine STATUS in the debugger, you probably want to examine it as an integer value. By default, the debugger uses the program-assigned data types when it displays program locations.

However, if necessary, you can specify a data type for a program location other than the data type your program has associated with it.

- SET TYPE/OVERRIDE command—Sets the default data type for debugger commands that interpret and display program data. For example, if you set the default data type to be BYTE, any variable you examine (regardless of how you declared it in your program) will be displayed as a BYTE value. The first EXAMINE command in the following example displays ADULTS as a REAL*4 value because the program unit CALC_SUMS declared ADULTS as a REAL*4 value; the second EXAMINE command displays ADULTS as a BYTE value because of the SET TYPE/OVERRIDE command.

  ```
  DBG> EXAMINE ADULTS
  CALC_SUMS\ADULTS:   1.000000
  DBG> SET TYPE/OVERRIDE BYTE
  DBG> EXAMINE ADULTS
  CALC_SUMS\ADULTS:   -128
  ```

- Data type qualifiers—Indicates that the modified command should display or evaluate the referenced location using the data type specified by the qualifier. A type qualifier overrides the default type specified with the SET TYPE/OVERRIDE command. In the following example, the default type for the debugging session is set to BYTE. The EXAMINE command uses the /FLOAT qualifier (FORTRAN data type REAL*4) to examine the variable ADULTS.

```
DBG> SET TYPE/OVERRIDE BYTE
DBG> EXAMINE ADULTS
CALC_SUMS\ADULTS:  -128
DBG> EXAMINE/FLOAT ADULTS
CALC_SUMS\ADULTS:  1.000000
```

The following table displays debugger data types and their FORTRAN equivalents (other languages may have different data types available, see your language-specific programming manual). The SHOW TYPE command displays the default FORTRAN data type, and the SHOW TYPE/OVERRIDE command displays the current /OVERRIDE type setting.

Debugger	FORTRAN
BYTE	LOGICAL*1
WORD	INTEGER*2, LOGICAL*2
LONG	INTEGER*4, LOGICAL*4
FLOAT	REAL*4
D_FLOAT	REAL*8
G_FLOAT	REAL*8
H_FLOAT	REAL*16
ASCII[:n]	CHARACTER*n

In addition to the familiar data types, the FORTRAN debugger provides an INSTRUCTION data type, which interprets values as VAX machine code instructions. The INSTRUCTION data type is a powerful debugging tool for those programmers who are familiar with machine code instructions.

5.6.5 Specifying Radix

To specify a radix other than the decimal default, use either the SET MODE command or include a radix qualifier with individual debugger commands.

- SET MODE command—Sets the default radix and symbolic mode for debugger commands that display and interpret data. The default for FORTRAN is decimal radix and symbolic mode (displaying symbolic rather than numeric addresses). For example, the following commands display the value of ADULTS before and after establishing hexadecimal as the default radix mode.

  ```
  DBG> EXAMINE ADULTS
  CALC_SUMS\ADULTS:     10
  DBG> SET MODE OCTAL
  DBG> EXAMINE ADULTS
  CALC_SUMS\ADULTS:     012
  ```

- Radix qualifier—Controls the radix of values interpreted by individual commands (and whether those commands display symbolic or numeric addresses). A qualifier that specifies radix overrides the default radix set by the SET MODE command, as is shown in the following example:

  ```
  DBG> EXAMINE ADULTS
  CALC_SUMS\ADULTS:      100
  DBG> EXAMINE/HEX ADULTS
  CALC_SUMS\ADULTS:      64
  ```

To display the current type and mode settings, use the SHOW TYPE and SHOW MODE commands. For example, the following commands display the FORTRAN default type and mode.

```
DBG> SHOW TYPE
type: long integer
DBG> SHOW MODE
mode:  symbolic, noscreen, keypad
input radix:  decimal
output radix: decimal
```

5.7 Using Command Procedures

You can create a command procedure that executes a sequence of debugger commands when you specify the at sign with the file's specification. For example:

```
DBG> @DEBUG
```

You can execute a command procedure interactively, from within a DO command sequence, or from within another command procedure. Command procedures are especially useful when you regularly perform a number of standard set up debugger commands; see Section 5.7.3 for information about initialization files. The following command procedure includes several SET commands as well as a call to another command procedure, ACTBUG.COM.

DEBUG.COM

```
SET MODULE/ALL
SET STEP SOURCE,INTO
SET LOG ACCTS.LOG
SET OUTPUT LOG,VERIFY
@ACTBUG
```

5.7.1 Displaying Commands

To display the commands in a command procedure (or DO command sequence) as they execute, specify the VERIFY keyword of the SET OUTPUT command.

```
DBG> SET OUTPUT VERIFY
```

For instance, the following commands show the effect of the SET OUTPUT VERIFY command upon the execution of the command procedure DEBUG.COM. The debugger reports entrance into and exit from nested command procedures.

```
DBG> SET OUTPUT VERIFY
DBG> @DEBUG
 SET MODULE/ALL
 SET MODE SYMBOL
 SET STEP SOURCE, INTO
 SET OUTPUT LOG,VERIFY
 SET LOG ACCTS.LOG
 @ACTBUG
%DEBUG-I-VERIFYICF, entering indirect command file "ACTBUG"
 SET SCOPE CALC_SUMS
 TYPE 1
MODULE CALC_SUMS
     1:          INTEGER FUNCTION CALC_SUMS (TOTAL_HOUSES,
%DEBUG-I-VERIFYICF, exiting indirect command file "ACTBUG"
%DEBUG-I-VERIFYICF, exiting indirect command file "DEBUG"
DBG>
```

5.7.2 Passing Values

To pass data to a command procedure:

1 Include a DECLARE command in the command procedure to associate a symbol with each parameter to be passed to the command procedure.

2 Specify the parameter values after the procedure name when invoking the command procedure.

For example, the following command procedure expects one parameter value, which will be associated with the symbol P1. The parameter value is then assigned to the program symbol J.

J.COM

```
SET SCOPE CALC_SUMS
SET BREAK CALC_SUMS
STEP 5
EXAMINE J
DECLARE P1:VALUE
DEPOSIT J = P1
EXAMINE J
GO
```

To pass the value 3 to J.COM, specify the following:

```
DBG> @J 3
stepped to INCOME\%LINE 177
   177:             READ (UNIT=STATS_LUN,
CALC_SUMS\J:          0
CALC_SUMS\J:          3
break at routine CALC_SUMS
DBG>
```

5.7.3 Initialization Files

You can use a command procedure as an initialization file by equating it to the logical name DBG$INIT. The file assigned the name DBG$INIT automatically executes at debugger start up. The following command procedure contains commands commonly used to set up for a debugging session.

DBGSTART.COM

```
! If source is not in my directory, use [INC.SOURCE]
SET SOURCE [],[INC.SOURCE]
SET MODULE/ALL
SET MODE SYMBOL,SCREEN,KEYPAD
SET STEP SILENT
SET OUTPUT LOG,VERIFY
SET LOG ACCTS.LOG
```

To make the file an initialization file, specify the DCL command DEFINE.

```
$ DEFINE DBG$INIT WORK:[USER]DBGSTART.COM
```

5.8 Using Log Files

A debugger log file maintains a record of each debugger command and display that occurs during a debugging session. The DBG> prompt is not recorded and the displays are automatically commented out with exclamation points to allow the use of a log file as a command procedure.

To create a log file, specify both of the following:

- SET OUTPUT—Directs debugger output to a log file (as well as the terminal)
- SET LOG—Specifies the name of the log file

For example, to create a log file named FIXINCOME.LOG, specify LOG as the parameter of the SET OUTPUT command and FIXINCOME.LOG as the parameter of the SET LOG command.

```
DBG> SET LOG FIXINCOME.LOG
DBG> SET OUTPUT LOG
```

The default file specification of a log file is DEBUG.LOG.

To use a log file as a command procedure, invoke it.

```
DBG> @FIXINCOME.LOG
%DEBUG-I-VERIFYICF, entering indirect command file "FIXINCOME.LOG"
 !%DEBUG-I-VERIFYICF, exiting indirect command file "DBG$INIT"
 SET BREAK CALC_SUMS\%LINE 35
              .
              .
              .
 EXIT
%DEBUG-I-VERIFYICF, exiting indirect command file "FIXINCOME.LOG"
DBG>
```

6 Data Structures

You can define data items for use by your program and associate these items with symbolic names to reference them. In general, data items are numeric (integers, real numbers, complex numbers), character, or logical in type. In addition, integers can be treated as logical or untyped data. This chapter describes the FORTRAN data structures; see your programming manual for the data structures available in other languages.

6.1 Definition and Reference of Data Items

Defining a data item means creating the entity and assigning its characteristics. Referencing a data item means using the entity. Data items are defined and referenced as follows:

- Variables—Data items that can change in value as the program executes are called variables. A type declaration statement explicitly defines a variable and takes the following form:

  ```
  data-type symbolic-name [,...]
  ```

  Programming in VAX FORTRAN describes the valid data types in FORTRAN. (See your language-specific programming manual for the valid data types available in your language.) The symbolic name consists of 1 through 31 alphabetic, numeric, underscore, or dollar sign characters). Unless you are declaring a system-defined symbol, the symbolic name is your own invention. (Most system symbols include a dollar sign; you can avoid accidentally redefining a system symbol by not using dollar signs in your symbol names.) You can define any number of variables of one type in one type declaration statement by separating the symbolic names with commas. The following FORTRAN example defines an integer variable with the symbolic name TOTAL_HOUSES.

  ```
  INTEGER TOTAL_HOUSES
  ```

  Type declaration statements must appear in the definition part of a program unit. In the execution part of a program unit, you can define variables implicitly by referencing an undefined symbolic name. By default, FORTRAN implicitly defines an integer variable for names beginning with the letters I, J, K, L, M, and N, and a real variable for names beginning with all other letters. (You can inhibit the implicit definition of variables or change the defaults with the IMPLICIT statement.) Assuming that I is not explicitly defined, the following example implicitly defines a variable named I as an integer.

```
DO I = 1, TOTAL_HOUSES
```

- Constants—Data items whose values are fixed are called constants. You define a constant implicitly by referencing its value. No symbolic name is associated with the constant. The right-hand side of the following example defines a constant with the value 1.

  ```
  STATUS = 1
  ```

  You can assign a symbolic name to a constant by using a PARAMETER statement, which must appear in the definition part of a program unit. The following FORTRAN statements define two INTEGER constants.

  ```
  INTEGER STATUS_OK,
  2       IO_OK
  PARAMETER (STATUS_OK = 1,
  2          IO_OK = 0)
  ```

  When you assign a symbolic name to a constant, a type declaration statement defines the data type of the constant. If you do not provide a type declaration statement, the constant takes an implicit data type based on the symbolic name.

- Function references—When a function executes, it replaces its invocation with a value. The intrinsic function SQRT, for example, evaluates to the square root of the argument specified in its invocation. The following statement fragment means 9.0.

  ```
  SQRT (81.0)
  ```

  Except for FORTRAN intrinsic functions, you must specify the data type of a function in a type declaration statement, as you do a variable or constant, or accept the implicit data types. The data types for the FORTRAN implicit functions are listed in *Programming in VAX FORTRAN*.

- Expressions— Expressions are values combined by operators. The values can be any combination of variables, constants, function references, or other expressions. The following statement fragment joins the values of a variable and a constant with the addition operator.

  ```
  TOTAL_HOUSES + 1
  ```

  The values being joined must all be of the same type or be convertible to the same type—for example, you cannot add a character value and a numeric value. At program execution time, an expression evaluates to a single value.

6.2 Assignment of Values to Variables

The assignment statement, which can only appear in the execution part of a program unit, assigns a value to a variable. It takes the following general form:

```
variable = value
```

The value (the right-hand side of the assignment statement) can be a variable, constant, function reference, or expression. The variable and the value must be of the same data type or the value must be convertible to the variable data type (see Section 6.8). At execution time, the variable takes on the value of the right-hand side of the assignment statement.

Examples

```
TOTAL_HOUSES = 1
```

Assigns the integer value 1 to TOTAL_HOUSES.

```
TOTAL_HOUSES = TOTAL_HOUSES + 1
```

Increments the value of TOTAL_HOUSES by 1.

```
AVG_PERSONS_HOUSE = PERSONS / TOTAL_HOUSES
```

Assigns the value of the variable PERSONS divided by the value of the variable TOTAL_HOUSES to AVG_PERSONS_HOUSE.

```
ROOT = SQRT (BASE)
```

Assigns the square root of the value of the variable BASE to ROOT.

In FORTRAN, variables are not automatically initialized. (Static variables are initially 0; however, you should not depend on these zero values.) You can initialize a variable by placing a constant enclosed in slashes (/) immediately after the name of the variable in the type declaration statement. The following example initializes the variable LINES to 1.

```
INTEGER LINES /1/
```

Alternatively, you can initialize a variable by naming the variable in a FORTRAN DATA statement, which must appear in the definition part of the FORTRAN program unit, as shown:

```
INTEGER LINES
DATA LINES /1/
```

Data initialization occurs only once—at the start of the program. In particular, local variables in subprograms are not initialized each time the subprogram is invoked; these variables retain their values from the previous invocation. In addition, a FORTRAN subprogram cannot use a DATA statement (or the slash notation) to initialize a variable that is passed to it as an argument or that is in a common block defined by another program unit. See Chapter 2 for information on local variables and Chapter 1 for information on passing data between program units.

6.3 Numeric Data

A numeric data item is an entity of 1, 2, 4, 8, or 16 bytes treated as a single number. Data items of type LOGICAL can also be manipulated as numeric data; see Section 6.5.

6.3.1 Bytes

The BYTE data type defines a variable as consisting of one byte of storage. You can use a byte variable as an integer containing values in the range of −128 through 127, or as a logical value (equivalent to LOGICAL*1).

6.3.2 Integers

An integer is a positive or negative whole number. The integer data types are as follows.

- INTEGER*4—Defines an integer that can have values in the range −2,147,483,648 through 2,147,483,647. An INTEGER*4 value takes four bytes of storage.
- INTEGER*2—Defines an integer that can have values in the range −32,768 through 32,767. An INTEGER*2 value takes two bytes of storage.
- INTEGER—Defers the typing decision to compile time. If the program unit is compiled (FORTRAN command) with the /I4 qualifier (the default), INTEGER means INTEGER*4. If the program unit is compiled with the /NOI4 qualifier, INTEGER means INTEGER*2.

Specify an integer constant as a whole number (no decimal points) optionally preceded by a plus or minus sign. An unsigned number is assumed to be positive. The following example defines the symbolic name TOTAL_HOUSES to mean an integer and assigns it the value of the integer constant 1.

```
! Definition part of program
INTEGER TOTAL_HOUSES
! Execution part of program
TOTAL_HOUSES = 1
```

To define integers of greater than four bytes (for example, a quadword integer for storing system time), use an array of integers as shown in Section 6.11.2.

6.3.3 Real Numbers

A real number is a positive or negative number with a decimal point or exponent. Machine code instructions for manipulating real data are implemented in hardware or emulated by software depending on your hardware.

The real data types are as follows

- REAL and REAL*4—Defines a single-precision real number that can have very low and high values (.29 times 10 raised to the power of –38 through 1.7 times 10 raised to the power of 38) but is precise only to approximately seven digits. Values exceeding seven digits in size are rounded. A REAL*4 value is stored in F_floating format taking four bytes of storage.
- REAL*8 and DOUBLE PRECISION—Defines a double-precision real number in either D_floating format (default) or G_floating format (if /G_FLOATING is specified at compile time). A double-precision number in D_floating format has the same range of values as a single-precision number (REAL*4) but much greater precision—approximately 16 digits. A double-precision number in G_floating format has greater range (.56 times 10 raised to the power of –308 through .9 times 10 raised to the power of 308) but less precision (approximately 15 digits) than a double-precision number in D_floating format. A REAL*8 value is stored in D_floating or G_floating format taking eight bytes of storage.
- REAL*16—Defines a quad-precision number that can have values in the range of .84 times 10 raised to the power of –4932 through .9 times 10 raised to the power of 4932, with an approximate precision of 33 digits. A REAL*16 value is stored in H_floating format taking 16 bytes of storage.

Specify a real constant as follows:

```
[sign]multiplierE[sign]exponent --- REAL*4
[sign]multiplierD[sign]exponent --- REAL*8
[sign]multiplierQ[sign]exponent --- REAL*16
```

The multiplier can be a whole number or a number that includes a decimal point. The exponent must be a whole number. The value of the constant is the multiplier times 10 raised to the power of the exponent. For example, 2.1E3 means 2.1 times 10 cubed, or 2100. You can precede the entire constant by a plus sign (default) or minus sign to indicate a positive or negative number. For example, –2.1E3 means –2100. You can precede the exponent by a plus sign (default) or minus sign to indicate a positive or negative power of 10. For example, 2.1E–3 means 2.1 times 10 to the power of –3, or .0021.

You can also specify a REAL*4 constant simply as a number, optionally preceded by a plus or minus sign, that has a decimal point. Be sure to include the decimal point even if the number is not fractional—for example, specify 1 as 1.0. A whole number is an integer constant. An unsigned number is assumed to be positive. The following example defines the symbolic name ROOT_ADD_1 as a real number (F_floating format) and assigns it the value of the square root of 81 plus 1, specifying the constants as real numbers.

```
! Definition part of program
REAL ROOT_ADD_1
! Execution part of program
ROOT_ADD_1 = SQRT (81.0) + 1.0
```

6.3.4 Complex Numbers

A complex number consists of a real part and an imaginary part. The complex data types are as follows

- COMPLEX and COMPLEX*8—Defines a complex number using single-precision (REAL*4) numbers for both its real and imaginary parts. A COMPLEX*8 value is stored as two single-precision numbers in F_floating format taking eight bytes of storage.
- COMPLEX*16—Defines the complex number using double-precision (REAL*8) numbers for both its real and imaginary parts. A COMPLEX*16 value is stored as two double-precision numbers in either D_floating (default) or G_floating (if /G_FLOATING is specified at compile time) format taking 16 bytes of storage.

Specify a complex constant as follows:

```
(real,imaginary)
```

The real and imaginary components can be either integer or real constants in any combination, except: inclusion of a REAL*8 constant as either component defines a COMPLEX*16 constant rather than a COMPLEX*8 constant; and inclusion of a REAL*16 constant as either component is illegal.

For example, (1.3,–1) defines a COMPLEX*8 constant, while (1.3D0,–1) defines a COMPLEX*16 constant.

6.4 Numeric Operations

Numeric expressions permit you to combine and compare numbers with special operators. In addition, system-defined procedures and FORTRAN intrinsic functions permit more involved operations.

6.4.1 Arithmetic Operations

FORTRAN provides operators for the basic arithmetic operations of exponentiation (**), multiplication (*), division (/), addition (+), and subtraction (–). You specify the operation by placing the operator between two values in an expression. (You also use the plus and minus signs as unary operators to indicate positive and negative numbers.) The values can be numeric variables, constants, function references, or expressions.

You can specify any number of operations in one expression. The operations are performed in the normal order of arithmetic evaluation: exponentiation first, followed by multiplication and division, followed by addition and subtraction. Operations enclosed in parentheses are performed first. Parenthesized operations can be nested; the innermost operations are performed first. Operations of equal value are performed left to right, with the exception of exponentiation which is performed right to left.

Note the following examples of arithmetic operations.

Examples

```
TOTAL_HOUSES + 1
```

Increments the value of TOTAL_HOUSES by 1.

```
PERSONS (I) / TOTAL_HOUSES
```

Divides the value of element I in the array PERSONS by the value of TOTAL_HOUSES.

```
SQRT (J) * I + 1
```

Multiplies the square root of J (SQRT (J) is a function reference) by the value of I and adds 1.

```
SQRT (J) * (I + 1)
```

Multiplies the square root of J by the sum of the value of I and 1.

```
3 + 3**2 + 4 * 2
```

Evaluates to 20: $3^2 = 9$, $4 * 2 = 8$, $3 + 9 + 8 = 20$.

```
2**3**2
```

Evaluates to 512: $3^2 = 9$, $2^9 = 512$.

```
(3 + 3)**2 + 4 * 2
```

Evaluates to 44: $3 + 3 = 6$, $6^2 = 36$, $4 * 2 = 8$, $36 + 8 = 44$.

```
3 + 3**(2 + 4) * 2
```

Evaluates to 1461: $2 + 4 = 6$, $3^6 = 729$, $729 * 2 = 1458$, $1458 + 3 = 1461$.

```
(3 + (3**2 + 4)) * 2
```

Evaluates to 32: $3^2 = 9$, $9 + 4 = 13$, $13 + 3 = 16$, $16 * 2 = 32$.

6.4.2 Relational Operations

FORTRAN provides relational operators for comparing values for equality (.EQ.), inequality (.NE.), less than (.LT.), less than or equal (.LE.), greater than (.GT.), and greater than or equal (.GE.). Comparisons are based on numeric value. You specify the operation by placing the operator between two values. The values can be variables, constants, function references, or expressions. The result of a comparative operation is a logical value of true (the relation is true) or false (the relation is false).

You can mix relational and arithmetic operations in an expression. Relational operations rank below arithmetic operations in order of evaluation so that typically you are comparing those items of the expression on either side of the relational operator. (You can change the order of evaluation with parentheses.)

Note the following examples of relational operations.

Examples

```
TOTAL_HOUSES .EQ. 0
```

True if the value of TOTAL_HOUSES equals 0.

```
IOSTAT .NE. IO_OK
```

True if the value of IOSTAT does not equal IO_OK.

```
HOUSE_NO .GT. MAX_STATS
```

True if the value of HOUSE_NO is greater than the value of MAX_STATS.

```
HOUSE_NO + 1 .GT. MAX_STATS
```

True if the sum of the value of HOUSE_NO and 1 is greater than the value of MAX_STATS.

```
3**2 .EQ. 3*3
```

True.

6.4.3 System Arithmetic Routines

The following intrinsic functions perform arithmetic operations.

ABS	Absolute value
ACOS	Arc cosine
ASIN	Arc sine
ATAN	Arc tangent
ATAN2	Arc tangent a1/a2
COS	Cosine
COSH	Hyperbolic cosine
DIM	Positive difference
EXP	Exponential
LOG	Natural logarithm
LOG10	Common logarithm
MAX	Maximum value of a list
MIN	Minimum value of a list
MOD	Remainder
SIGN	Transfer of sign
SIN	Sine
SINH	Hyperbolic sine
SQRT	Square root
TAN	Tangent
TANH	Hyperbolic tangent

Use the intrinsic subroutine RAN to obtain a random number.

The FORTRAN data types permit you to apply arithmetic operations to integers of one byte (using the BYTE or LOGICAL*1 data type), one word (INTEGER*2), or one longword (INTEGER*4). To add and subtract integers of one quadword or other lengths, use the Run-Time Library procedures LIB$ADDX and LIB$SUBX. Specify a quadword as an array of two INTEGER*4 integers. See Section 6.11.2.2, which describes manipulating time, for an example.

6.4.4 Arithmetic Errors

The system detects and signals the following arithmetic errors at program run time (where constants are used, some of these errors may be trapped as compile-time errors).

Error	Symbol
Integer overflow	SS$_INTOVF
Integer divide by 0	SS$_INTDIV
Floating overflow	SS$_FLTOVF_F
Floating underflow	SS$_FLTUND_F
Floating divide by 0	SS$_FLTDIV_F
Decimal overflow	SS$_DECOVF
Unspecified	SS$_ARTRES

You can control the signaling of integer overflow errors and floating-point underflow errors by specifying either (or both) the /CHECK=[NO]OVERFLOW or /CHECK=[NO]UNDERFLOW qualifiers at compile time. By default, integer overflow errors are signaled but floating-point underflow errors are not.

6.5 Logical Data

A logical data item is an entity of one, two, or four bytes treated as a value of true or false. The item has a value of true if its value as an integer is odd—that is, the low-order bit is on; a value of false if its value as an integer is 0 or even—that is, the low-order bit is off. The logical data types are as follows:

- LOGICAL*1—Defines the logical item as taking one byte of storage.
- LOGICAL*2—Defines the logical item as taking two bytes of storage.
- LOGICAL*4—Defines the logical item as taking four bytes of storage.
- LOGICAL—Defers the typing decision to compile time. If the program unit is compiled (FORTRAN command) with the /I4 qualifier (the default), LOGICAL means LOGICAL*4. If the program unit is compiled with the /NOI4 qualifier, LOGICAL means LOGICAL*2.

In general, the integer and logical data types are interchangeable: a data item defined as an integer can be operated on as a logical item and the reverse.

You can specify logical values as follows:

- Constants—The special constant .TRUE. means true; numerically, .TRUE. evaluates to –1 (all bits are on). The special constant .FALSE. means false; numerically, .FALSE. evaluates to 0 (all bits are off). The following example assigns a value of true to a variable.

  ```
  ! Definition statements
  LOGICAL READ_ONLY
  ! Execution statements
  READ_ONLY = .TRUE.
  ```

- Numbers—An odd integer value means true, and 0 or an even integer value means false. The following example assigns a logical value of true to a variable.

  ```
  ! Definition statements
  INTEGER STATUS
  ! Execution statements
  STATUS = 1
  ```

- Relations—A compare operation with a true logical value (the relationship is true) has a numeric value of –1. A compare operation with a false logical value (the relationship is false) has a numeric value of 0. The following example assigns a value of true to STATUS if IOSTAT equals IO_OK and a value of false if IOSTAT does not equal IO_OK.

  ```
  STATUS = IOSTAT .EQ. IO_OK
  ```

- Logical operators—The special operators .NOT., .AND., .OR., .XOR., .NEQV., and .EQV. operators are ranked below the comparison operators in precedence of evaluation. The following example makes STATUS true if IOSTAT is false, false if IOSTAT is true.

  ```
  STATUS = .NOT. IOSTAT
  ```

The IF and DO WHILE statements test a logical value to decide whether to execute or not execute a block of statements that follow. (See Chapter 2 for a discussion of IF and DO logic.) A true value means a block is executed, a false value means it is skipped. The following example executes the block of statements if the value of STATUS is true.

```
IF (STATUS) THEN
  <block of statements>
END IF
```

The logical value being tested can be any logical or integer expression, as demonstrated:

```
IF ((STATUS) .AND. (IOSTAT .EQ. IO_OK))
```

In this statement, the system evaluates STATUS, compares IOSTAT and IO_OK, and then combines the results of the two operations using the .AND. logical operator. The parentheses around STATUS and IOSTAT .EQ. IO_OK are not really necessary, as the comparison operator .EQ. ranks higher than the logical operator .AND., but parentheses help clarify the source code.

At times it is possible for FORTRAN to determine the result of an expression before completing evaluation of all subexpressions. In the previous example, if STATUS is false, the subexpression (IOSTAT .EQ. IO_OK) is not evaluated. Do not depend on the side effects of a function invoked within a logical expression since that function may not be executed.

6.6 Character Data

A character string is a series of one or more characters. Each character in a string is an entity of one byte interpreted according to ASCII conventions .

6.6.1 Defining Character Strings

The type declaration statement for a character string takes the following form:

```
CHARACTER*size symbolic-name
```

All character strings in FORTRAN are fixed-length strings. **Size** represents the number of characters in the string and must be specified as an integer constant. The following example defines a character string of 12 characters.

```
CHARACTER*12 FIX_HOUSE_STRING
```

Assumed-size character strings permit you to omit the size of the character string. You can use assumed-size character strings in the following cases:

- Dummy argument—When the character string is a dummy argument, **size** is the length of the actual argument.
- PARAMETER constant—When the character string is a constant defined in a PARAMETER statement, **size** is the length of the specified constant.

Specify an assumed-size character string by substituting an asterisk in parentheses for the size of the array. In the following example, the character string INSTRUCTIONS assumes the size of the character constant specified in the PARAMETER statement.

```
CHARACTER*(*) INSTRUCTIONS
PARAMETER (INSTRUCTIONS = 'Enter a statistic and hit RETURN')
```

6.6.2 Character Constants

Specify a character constant as a series of printable characters enclosed in apostrophes. To include an apostrophe in the constant, type two consecutive apostrophes.

```
STRING = 'I said it couldn''t be true.'
```

6.6.3 Referencing Character Strings

Reference a character string variable by specifying the variable name in a context that allows a CHARACTER data type. To reference a part of a character string (called a substring), use a reference of the following form:

```
symbolic-name ([first]:[last])
```

State the first and last positions of the substring within the string, in parentheses separated by a colon. The positions can be specified as integer constants, variables, function references, or expressions. First defaults to the first character in the string and last defaults to the last character in the string. The following example writes the number of characters in STRING as specified by the integer variable SIZE.

```
! Definition statements
CHARACTER*12 STRING
INTEGER SIZE
! Execution statements
TYPE *, STRING (1:SIZE)
```

Take care to reference substrings of one character in the proper format. The first character of STRING, for example, is STRING (1:1), not STRING (1).

6.6.4 Character String Operations

You should reference only the defined portion of a character string. For instance, if STRING is a 12-character string and you have assigned it a value of less than 12 characters, reference the significant characters using a substring. The following example writes the defined portion of STRING (the number of characters as specified by the INTEGER variable SIZE) to SYS$OUTPUT.

```
TYPE *, STRING (1:SIZE)
```

Where you are building the character value, you should calculate and maintain its size in an integer variable. If you are obtaining the character value (for example, through an I/O operation or a Run-Time Library procedure), you may or may not be supplied its length. If you are supplied the length (for example, as a return argument), you should save it in an integer variable. If you are not supplied the length, you should calculate it, for example, by using the system-defined procedure STR$TRIM to return the length of the string minus any trailing blanks or tabs. The following example writes the length of the value in STRING to SIZE.

```
! Definition statements
CHARACTER*12 STRING
INTEGER SIZE
! Declare status and system routines
INTEGER STATUS,
2       STR$TRIM
! Execution statements
STATUS = STR$TRIM (STRING,  ! Destination
2                  STRING,  ! Source
2                  SIZE)    ! Length
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

6.6.4.1 Padding and Truncation

If you assign a character string to a variable and the length of the character string is less than the length defined for the variable, the character string is left-justified and the variable is padded on the right with blanks. The string lengths returned by system-defined procedures do not count the padding blanks.

If you assign a character string to a variable and the length of the character string is greater than the length defined for the variable, the character string is left-justified and truncated on the right. The string lengths returned by system-defined procedures do not count the truncated characters.

6.6.4.2 Concatenation

The concatenation operator (//) combines two character strings so that the last character of the first string adjoins the first character of the second string. Remember to trim unwanted blanks from both character strings, especially the first, before joining them. The following example concatenates a character constant and variable to form the first argument of a function reference.

```
TYPE *, 'Average income per household: $' // STRING (1:SIZE)
```

6.6.4.3 Intrinsic Character Functions

The following intrinsic functions operate on character strings.

LEN	Determines the length of a character string
INDEX	Determines the starting position of a substring
CHAR	Returns a character whose ASCII value is as specified in the integer argument
ICHAR	Returns an integer whose value is the ASCII value of the character argument

6.6.5 Nonprintable Characters

A number of ASCII characters do not have corresponding terminal keys and printing characters. You can represent a nonprintable character by specifying its ASCII value as an argument to the intrinsic function CHAR. The argument is an integer value and CHAR returns a one-character constant. The following example rings the bell on the terminal (the ASCII character represented by the decimal value 7 is the control character BEL).

```
CHARACTER*1 BEL
PARAMETER  (BEL = CHAR(7))
TYPE *, BEL
```

You can concatenate nonprintable characters with other characters using the concatenation character (//). The following example rings the bell before displaying a line on the terminal.

```
CHARACTER*1 BEL
PARAMETER (BEL = CHAR(7))
CHARACTER*(*) IMP
PARAMETER (IMP = BEL//'IMPORTANT NOTICE FOLLOWS')
TYPE *, IMP
```

6.6.6 Counted Strings

System-defined procedures may pass you data or require data in the form of a *counted string*. A counted string is a character string or array of no more than 256 characters. The first character is the character representation of a binary count of the remaining characters. The following example sets up a counted string from data read from SYS$INPUT.

```
INTEGER COUNT
CHARACTER*256 STRING
READ (UNIT=*,
2     FMT='(Q,A)') COUNT,
2                  STRING (2:256)
STRING (1:1) = CHAR (COUNT)
```

Do not convert the count from or to a BYTE (or LOGICAL*1) data type because the upper positive value is only 127. Use at least an INTEGER*2 data type. The following example displays the contents of a counted string.

```
INTEGER COUNT
CHARACTER*256 STRING
COUNT = ICHAR (STRING (1:1))
WRITE (UNIT=*,
2      FMT='(A)') STRING (2:COUNT+1)
```

6.7 Untyped Data

In general, you should try to work within the numeric and character data types. The capabilities exist, however, to define and manipulate data as bit configurations.

6.7.1 Untyped Constants

You can treat hexadecimal, octal, and Hollerith constants as numeric (integer, real, complex, logical, and byte) data. Character constants may also be used as numeric data; however, a character constant in a numeric context is considered a Hollerith constant with the length of the specified character constant.

An untyped constant has a value corresponding to its bit configuration. An untyped constant assumes a data type based on its context.

- Expressions—When used with a binary operator (an operator that joins two elements; includes the assignment operator), the constant assumes the data type of the other operand.

  ```
  INTEGER*2 WORD
  INTEGER*4 LONG
  REAL*8    DOUBLE

  IF (LONG .LE. '123'O) THEN      ! Constant is INTEGER*4
    LONG = WORD + '123'O          ! Constant is INTEGER*2
    DOUBLE = '123'O               ! Constant is REAL*8
  END IF
  ```

- Required data types—When used in a context that requires a particular data type (generally integer), the constant assumes the required data type.

  ```
  REAL*8 DOUBLE,
  2      ARRAY(20)

  DOUBLE = ARRAY('16'O) + 3.5     ! Constant is INTEGER*4
  ```

- Actual arguments—When used as an actual argument, the constant does not assume a data type. For hexadecimal and octal constants, FORTRAN passes exactly four bytes. For Hollerith constants, FORTRAN passes the constant: if the constant is shorter than the dummy argument, it is padded with zero bytes; if the constant is longer than the dummy argument, it is truncated.

When used in any other context, an untyped constant assumes an INTEGER*4 data type.

6.7.1.1 Hexadecimal Constants

Specify a hexadecimal constant as a series of digits in the range 0 through 9 and letters in the range A through F (uppercase or lowercase), enclosed in apostrophes and followed by the letter X (in uppercase or lowercase). Each hexadecimal character represents a configuration of four bits with binary and decimal values as follows:

Hex	Binary	Decimal	Hex	Binary	Decimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

Two hexadecimal characters represent precisely one byte. The rightmost character represents the low-order four bits and the leftmost character represents the high-order four bits. In the following example, the variable HEX is assigned a binary value of 01000010 (decimal 66).

```
BYTE HEX
HEX = '42'X
```

You can specify up to 32 hexadecimal characters in one constant, equivalent to 128 bits or 16 bytes. If the hexadecimal constant contains fewer digits than fill the variable, the variable is padded on the left side with zeros. If the hexadecimal constant contains more digits than the variable can hold, the constant is truncated on the left. In the case of truncation, you receive warning messages at compile and link time; however, the program does compile and link with the expected results.

6.7.1.2 Octal Constants

Specify an octal constant as a series of digits in the range 0 through 7, enclosed in apostrophes and followed by the letter O (in uppercase or lowercase). Each octal character represents a configuration of three bits with binary and decimal values as follows:

Octal	Binary	Decimal	Octal	Binary	Decimal
0	000	0	4	100	4
1	001	1	5	101	5
2	010	2	6	110	6
3	011	3	7	111	7

Octal constants are assigned to variables with the rightmost character representing the low-order three bits of the variable, the next character representing the next three bits, and so on. Octal constants do not fit into bytes precisely. Three octal constants, for example, represent nine bits, or one byte with one bit left over. Depending on the size of the variable to which the constant is being assigned, the leftover bit becomes the low-order bit of the next byte in the variable. In the following example, OCTAL1 is assigned a binary value of 00111111 (decimal 63), while the low-order byte of OCTAL2 is assigned a binary value of 11111111, and the high-order byte is assigned a value of 1.

```
BYTE OCTAL1
INTEGER*2 OCTAL2
OCTAL1 = '77'O
OCTAL2 = '777'O
```

You can specify up to 43 octal characters in one constant, equivalent to 128 bits or 16 bytes (but the high-order character of a 43-character specification cannot exceed a value of 6). If the octal constant contains fewer digits than fill the variable, the variable is padded on the left side with zeros. If the octal constant contains more digits than the variable can hold, the constant is truncated on the left. In the case of truncation, you receive warning messages at compile and link time; however, the program does compile and link with the expected results.

6.7.1.3 Hollerith Constants

Specify a Hollerith constant as an integer, followed by the letter H (uppercase or lowercase), followed by a sequence of printable characters—the number of characters in the sequence must equal the value of the integer. Each character following the H corresponds to one byte in the numeric variable to which it is assigned; the variable takes the ASCII value of the character. (You cannot assign Hollerith constants to character variables.) In the following example, the variable LETTER is assigned the decimal value 66, the ASCII value of an uppercase B.

```
BYTE LETTER
LETTER = 1HB
```

The leftmost character of a Hollerith constant corresponds to the low-order byte of the numeric variable to which it is assigned. In the following example, the low-order byte of LETTERS takes a value of 65 and the high-order byte takes a value of 66 (the numeric value of LETTERS as an integer is 16,961—66 times 256 plus 65).

```
INTEGER*2 LETTERS
LETTERS = 2HAB
```

You can specify up to 2000 characters in a Hollerith constant. If the constant contains fewer characters than the variable contains bytes, the variable is padded on the right side with spaces (a space is a decimal 32 in ASCII). If the constant contains more characters than the variable contains bytes, the constant is truncated on the right. In the case of truncation, you receive warning messages at compile and link time; however, the program does compile and link with the expected results.

6.7.2 Bit Manipulation

To manipulate bits (for example, to set or examine mask values as described in Sections 1.5 and 1.6), use the intrinsic functions IAND, IOR, IEOR, NOT, ISHFT, IBITS, IBSET, BTEST, IBCLR, and ISHFTC. Note that these functions require that bits be referenced in integers (INTEGER, LOGICAL, and BYTE data types). In referring to the position of a bit, 0 means the low-order bit in the integer, 1 means the next bit, and so on. The following example sets the low-order and high-order bits in a longword.

```
INTEGER*4 MASK
MASK = IBSET (MASK, 0)
MASK = IBSET (MASK, 31)
```

6.8 Conversion Between Data Types

Conversion of values between data types is necessary internally when values are assigned to variables of different types, or values of different types appear in the same operation.

When moving data between internal storage and external units (such as terminals, printers, and disk files), conversion is required when the internal data is noncharacter and the external unit is a character-oriented device (for example, a terminal or printer) or a formatted file. Chapter 8 discusses terminal I/O and Chapter 9 discusses file I/O.

6.8.1 Numeric Conversions

When you assign a numeric value to a numeric variable and the data types are different, the value is converted to the data type of the variable. In the following example, the integer value of PERSONS is converted to a real number.

```
REAL PERSONS_HOUSE (2048)
INTEGER PERSONS
         .
         .
         .
PERSONS_HOUSE (I) = PERSONS
```

When you use a binary operator (an operator that joins two elements) to combine numeric values of different data types, the value with the lower ranking data type is converted to the data type of the other operand before the operation is performed. The data types are ranked integer, real, and complex, low to high, with the smaller or less precise data types below the larger or more precise data types (for example, REAL*4 ranks below REAL*8). In the following example, WORD is converted to an INTEGER*4 value and added to LONG. The result is then converted to a REAL*4 value and added to SINGLE.

```
INTEGER*2 WORD
INTEGER*4 LONG
REAL*4    SINGLE
         .
         .
         .
TYPE *, WORD + LONG + SINGLE
```

In converting from an integer to a real number, a fraction (.0) is appended to the integer. In converting from a lower data type to a complex number, an imaginary part (0.0) is added to the number; if necessary, a fraction (.0) is added to the real part of the complex number. You do not lose any part of the number in converting upwards except that conversion of a very large integer to a REAL*4 real number results in a loss of precision. For example, conversion of the integer value 123,456,789 into a real number (REAL*4) yields 1.2345679E+08 (123,456,790).

In converting from a complex number to a lower data type, the imaginary part is truncated. In converting from a real number to an integer, the fraction is truncated. Conversion of a number to a smaller and/or less precise data type (for example, a REAL*8 real number to a REAL*4 real number) may result in an overflow error (if the number is too large for the receiving data type) or a loss of precision.

6.8.2 Formatted Conversions

You cannot manipulate a character string as a number even if the string has the appearance of a number. For example, 9 + '2' is an illegal expression. And you cannot use a number as a character value. You must convert the number to a character string or the character string to a number.

The preferred mechanism for performing character-to-numeric conversions is the internal READ statement. The preferred mechanism for performing numeric-to-character conversions is the internal WRITE statement. In internal READ and WRITE statements, the I/O unit is specified as the variable containing the character value (rather than as an external device or file) and the I/O list is the variable or constant containing the number. A format must be specified with the FMT specifier of the I/O statement.

The general form of an internal READ statement (for converting characters to numbers) is as follows:

```
READ (UNIT=character-variable,
2     FMT=format-spec) numeric-variable
```

The following example converts the character string in C to the integer N.

```
CHARACTER*10 C
INTEGER N
INTEGER*2 SIZE
CALL LIB$GET_INPUT (C,
2                   'Next number or END: ',
2                   SIZE)
READ (UNIT=C(1:SIZE),
2     FMT='(BN, I10)') N
```

The general form of the internal WRITE statement (for converting numbers to character strings) is as follows:

```
WRITE (UNIT=character-variable,
2      FMT=format-spec) numeric-value
```

The following example converts the integer in N to the character string C.

```
CHARACTER*10 C
INTEGER N,
2       SIZE
WRITE (UNIT=C,
2      FMT='(I10)') N
! Get rid of padding
SIZE = 1
DO WHILE ((C(SIZE:SIZE) .EQ. ' ') .AND. (SIZE .LE. 10))
  SIZE = SIZE + 1
END DO
! Display it
TYPE *,'Number = '//C(SIZE:10)
```

You can convert multiple values in one READ or WRITE statement by defining the character variables as an array. The following example converts two numbers to character strings.

```
CHARACTER*10 C (2)
INTEGER N1,
2       N2
WRITE (UNIT=C,
2      FMT='(2I10)') N1,
2                    N2
```

You can also use format specifications to convert data during regular I/O operations. For example, you can read directly from the terminal or a formatted file into numeric variables or write numeric values directly to the terminal or a formatted file. The following example writes an integer to the terminal.

```
INTEGER N
WRITE (UNIT=*,
2      FMT='(I4)') N
```

6.8.3 Automatic Conversions

You can also convert data during regular sequential I/O operations without specifying a format. In list-directed and namelist-directed I/O statements, conversion is automatic using the default formats for the data types of the values involved. The following example writes an integer to the terminal.

```
INTEGER N
TYPE *, N
```

6.9 Arrays

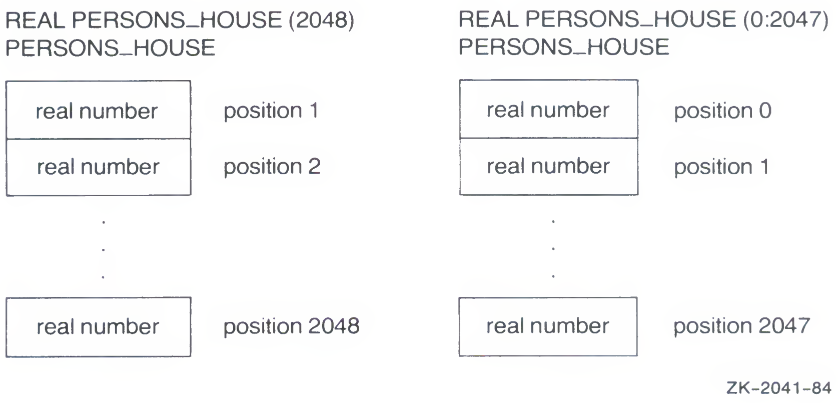
An array is a sequence of data items of the same type with one name. Each data item within the array is called an element and is associated with an integer value that designates the position of the element in the sequence. For clarity, Sections 6.9.1 through 6.9.3 refer only to one-dimensional arrays. Section 6.9.4 discusses multidimensional arrays.

6.9.1 Defining Arrays

You define an array with a type declaration statement of the following form:

```
data-type symbolic-name ([first:]last) [,...]
```

First designates the position of the first array element (lower bound) and **last** the position of the last array element (upper bound); the lower bound must be less than or equal to the upper bound. The increment between positions is always 1, so that the size of the array is **last** minus **first** plus 1. If you omit **first**, the array starts at position 1 and contains the number of elements specified by last. The following example demonstrates two array definitions, each defining an array of 2048 elements.



You can also define an array with the DIMENSION statement, which must appear in the definition part of a program unit.

```
DIMENSION symbolic-name ([first:]last) [,...]
```

You must declare the data type of the array elements explicitly with a type declaration statement or accept the implicit data type. The following example defines an array of 2048 real numbers.

```
REAL PERSONS_HOUSE
DIMENSION PERSONS_HOUSE (2048)
```

The values specifying the first and last positions of arrays in type and DIMENSION statements must be constants or expressions containing only constants or variables defined either in a common block or as dummy arguments (if the array is a dummy argument). See Chapter 1 for defining and using adjustable and assumed-size arrays as dummy arguments in subprograms.

6.9.2 Referencing Arrays

You reference a particular element in an array by appending a subscript to the name of the array. The subscript consists of a parenthesized integer value and designates the position of the array element. For example, PERSONS_HOUSE (5) means the element of PERSONS_HOUSE at position 5. The subscript value can be specified as a variable, constant, function reference, or expression. The use of variables as subscripts provides a very powerful tool for manipulating data, as demonstrated in the sections that follow.

In referencing an element of a character string array, place the element subscript before the substring specification. The following example places the first character of the fifth element of the character string array BOOK_NAME into the variable LETTER.

```
CHARACTER*31 BOOK_NAME (101)
CHARACTER LETTER
! Executable statements
LETTER = BOOK_NAME (5) (1:1)
```

6.9.2.1 Per-Element Processing

Typically you do not specify an array subscript as a constant, but calculate the position of the element you need. For example, if each element of the PERSONS_HOUSE array represents a statistic for one household, and the subscript corresponds to a number assigned to the household for identification purposes, you calculate the subscript by making it equal a particular household number. If you want to change the statistic for a particular household, you might require the user of your program to specify the household number. You read the household number and convert it to a numeric data type, placing it in the integer variable FIX_HOUSE_NO. You can now reference the required array element by using FIX_HOUSE_NO as the subscript (FIX_PERSONS_HOUSE is the new value being placed in the array element).

```
PERSONS_HOUSE (FIX_HOUSE_NO) = FIX_PERSONS_HOUSE
```

As another example, consider the initial entry of values into the array. The integer TOTAL_HOUSES represents the highest number household for which a statistic has been entered. The subscript for a new household, then, can be calculated as one greater than for the last (where PERSONS is the new value being added to the array).

```
TOTAL_HOUSES = TOTAL_HOUSES + 1
PERSONS_HOUSE (TOTAL_HOUSES) = PERSONS
```

Or

```
PERSONS_HOUSE (TOTAL_HOUSES + 1) = PERSONS
```

6.9.2.2 Multielement Processing

Typically you process the elements of an entire array or a part of an array with a DO loop. At the start of the first iteration of the loop, you make the subscript equal to the lowest (or highest) position of the array, then after each iteration of the loop you increment (or decrement) the subscript until you process the highest (or lowest) element of the array. In the following example, the control variable for the DO loop also serves as the subscript for the required array element. The example totals the array elements.

```
DO I = 1, TOTAL_HOUSES
  TOTAL_PERSONS = TOTAL_PERSONS + PERSONS_HOUSE (I)
END DO
```

In FORTRAN I/O statements, you can transfer an entire array by using an unsubscripted array name. However, if you have defined only part of an array, transfer the data by specifying the lower and upper bounds of the defined elements in an implied DO loop. An implied DO loop for processing an array takes the following form:

```
statement (control-list) (array-name (sub), sub = begin, end)
```

Sub is the control variable for the loop and the subscript for the variable. **Begin** and **end** designate the positions of that portion of the array to be processed. The following example writes that portion of PERSONS_HOUSE that contains statistics to a file.

```
WRITE (UNIT=STATS_LUN)
2     (PERSONS_HOUSE (I), I = 1, TOTAL_HOUSES)
```

In general, you should write arrays in this fashion to limit the transfer to that portion of the array containing useful values. For example, if PERSONS_HOUSE defines an array of 2048 elements but contains (at the time of the data transfer) only 921 significant elements (as reflected in the value of TOTAL_HOUSES), you do not want to transfer the entire array of 2048 elements. When reading data in this fashion, you must be precise. If the file record you are reading contains 921 elements of an array, you must transfer exactly 921 elements.

6.9.2.3 Full Array Processing

You can use array elements in the same contexts as variables of the same data type. The contexts in which you can use whole arrays are more limited. For example, whole arrays cannot appear as either the left-hand side or the right-hand side of an assignment statement. If you want to copy all the elements of one array into another array, you must do so element by element. You can specify whole arrays in the following contexts:

- COMMON—Specifies that the entire array is being placed in the common block. See Section 2.1.4 for restrictions on equivalencing arrays in common blocks.
- DATA—Specifies that every element of the array is being initialized. You must then specify a value for every element. The following example initializes every element of PERSONS_HOUSE to the numeric value 9999.

  ```
  REAL PERSONS_HOUSE (2048)
  DATA PERSONS_HOUSE /2048*9999/
  ```

  You can use an implied DO loop to initialize a portion of the array. The following example initializes the first 1024 elements of PERSONS_HOUSE.

  ```
  DATA (PERSONS_HOUSE (I), I = 1, 1024)
  2    /1024*99999/
  ```

- EQUIVALENCE—Associates two arrays with the same area of storage. See Section 2.1.4 for restrictions on equivalencing arrays in common blocks.
- FUNCTION—You can specify the entire array as a dummy argument.
- SUBROUTINE—You can specify the entire array as a dummy argument.
- ENTRY—You can specify the entire array as a dummy argument.
- SAVE—Refers to the entire array.

You can also use unsubscripted array names in I/O statements and as actual arguments in CALL statements and function references. If you have defined only a subset of the array elements, you may wish to use an implied DO loop (as shown in Section 6.9.2.2) when specifying the array in an I/O statement.

6.9.3 Storage and Bounds Considerations

In general, you should define larger arrays than you estimate you need (unless you are sure of the exact number of elements) so you do not run out of space. Do not worry about size unless you are close to your virtual memory limit (2MB; see Section 2.1.6). The storage you define is not really used until you reference it, and the transparent management of memory by the system is easier to use and usually more efficient than attempting to manage memory yourself (for example, by breaking an array into several records and processing it a record at a time).

When referencing an array, you should ensure that the array subscript is within the defined bounds. The following example returns an error status if the variable destined to be a subscript (HOUSE_NO) exceeds the array range (1:MAXSTATS).

```
REAL PERSONS_HOUSE (2048)
INTEGER HOUSE_NO
INTEGER MAXSTATS
PARAMETER (MAXSTATS = 2048)
EXTERNAL INCOME_MAXSTATS
     .
     .
     .
HOUSE_NO = HOUSE_NO + 1
IF ((HOUSE_NO .LT. 1) .OR.
2   (HOUSE_NO .GT. MAXSTATS)) THEN
  STATUS = %LOC (INCOME_MAXSTATS)
END IF
```

You can have the system trap out-of-range subscript references by specifying /CHECK=BOUNDS when you compile the program unit. When your program runs, the system signals the fatal error SS$_SUBRNG if a subscript reference is out of the array range. (An out-of-range subscript that is a constant is trapped at compile time.)

If you do not trap or let the system trap out-of-range references, results are unpredictable. You may access data or code in another portion of your program or attempt to enter an area of memory being used by another process causing an access violation.

6.9.4 Multidimensional Arrays

You can define arrays of up to seven dimensions. You specify multiple dimensions with multiple bounds specifiers in the data type declaration or DIMENSION statements, separating them with commas. The following example specifies an array of two dimensions. The bounds of the first dimension are positions 1 and 3; the bounds of the second dimension are positions 1 and 2048.

```
REAL STATS (3, 2048)
```

The total number of elements in a multidimensional array is the product of the number of elements in each dimension. The STATS array shown above has 6144 elements.

You reference an element of a multidimensional array by specifying a position in each dimension. The following example references the elements at position 958 (second dimension) of position 1 (first dimension) and position 958 (second dimension) of position 2 (first dimension).

```
<assume TOTAL_HOUSES equals 957>
TOTAL_HOUSES = TOTAL_HOUSES + 1
STATS (1, TOTAL_HOUSES) = PERSONS
STATS (2, TOTAL_HOUSES) = ADULTS
```

6.9.4.1 Storage of Multidimensional Arrays

Arrays are stored with the leftmost subscripts varying most rapidly. The example below demonstrates the absolute storage positions for an array with three positions in dimension 1 and 2048 positions in dimension 2.

Array Element	Storage Position
(1,1)	1
(2,1)	2
(3,1)	3
(1,2)	4
(2,2)	5
.	.
.	.
.	.
(2,2047)	6140
(3,2047)	6141

Array Element	Storage Position
(1,2048)	6142
(2,2048)	6143
(3,2048)	6144

If you think of a two-dimensional array as a table of columns (dimension 1) and rows (dimension 2), the order of storage would be all the elements in row 1, all the elements in row 2, and so on. If you think of a three-dimensional array as a number of tables on consecutive pages (columns, dimension 1; rows, 2; pages, 3), the order of storage would be all the elements on page 1 (stored row by row), all the elements on page 2 (stored row by row), and so on.

6.9.4.2 Processing Multidimensional Arrays

To process the elements of a multidimensional array, use a DO loop, similar to that described in Section 6.9.2.2, for each dimension. The nesting of the DO loops depends on the order in which you want the array elements processed. The innermost DO loop corresponds to the dimension varying the fastest; the outermost DO loop to the dimension varying most slowly. For more efficient programs, process the elements of multidimensional arrays in their order of storage.

For example, think of a three-dimensional array as described in Section 6.9.4.1: a series of pages (dimension 3) with a table of columns (dimension 1) and rows (dimension 2) on each page. To display the elements in each row in order beginning with the table on the first page, you would vary dimension 1 the fastest (to move across the row before beginning the next row), followed by dimension 2 (to move down the rows before beginning the next page), with dimension 3 varying most slowly. The following program segment displays each table, printing the page number at the beginning of each.

```
! Array
CHARACTER*3 CODES (10,20,6)
! Dimensions
INTEGER*4 MAX_PAGE /6/,
2         MAX_ROW /20/,
2         MAX_COLUMN /10/
```

```
! Physical page positions
INTEGER*4 LINE,
2         HEADER /2/,
2         LEFT_MARGIN /6/,
2         BEGIN_TABLE /5/
! Page number
CHARACTER*3 PAGE
            .
            .
            .
DO K = 1, MAX_PAGE
  ! At the beginning of each page print the page number
  ! Convert page number to character
  WRITE (UNIT = PAGE,
2        FMT = '(I4)') K
  STATUS = LIB$PUT_SCREEN ('Table - page '//PAGE, ! Text
2                          HEADER,                ! Line
2                          LEFT_MARGIN)           ! Margin
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  ! Line to begin table
  LINE = BEGIN_TABLE
  DO J = 1, MAX_ROW
    DO I = 1, MAX_COLUMN
      ! Leave 3 spaces between elements
      ! (3 spaces + 3 characters = 6 characters per element)
      STATUS = LIB$PUT_SCREEN (CODES (I,J,K),
2                              LINE,
2                              I*6)
    END DO     ! Columns
    ! At the end of each row update the line number
    LINE = LINE + 1
  END DO       ! Rows
END DO         ! Pages
            .
            .
            .
```

In FORTRAN I/O statements, transfer multidimensional arrays by nesting implied DO loops. A nested implied DO loop has the following form:

```
statement (control-list)
2  (...((array (a,b,...), a = begin,end), b = begin,end),...)
```

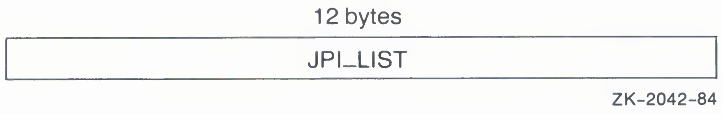
A, **b**, and so on are control variables for the loops and subscript values for the array variable. Each control variable has **begin** and **end** values that designate that portion of the dimension to be processed. The following example writes the three-dimensional array of the previous example to a file.

```
WRITE (UNIT=STATS_LUN)
2  (((CODES (I,J,K), I=1,MAX_COLUMN), J=1,MAX_ROW), K=1,MAX_PAGE)
```

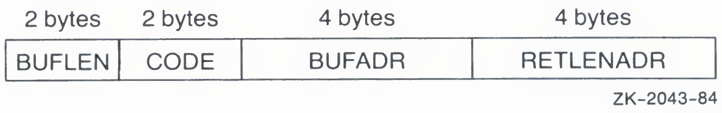
In general, you should limit the transfer to that portion of the array containing useful values. When transferring data in this fashion, you must be precise. You must read an array of exactly the same dimensions as the array you wrote.

6.10 Records

A record provides you with a data structure that can be addressed at several levels. At the top level, you address all the data in the record. At lower levels, you address fields and subfields within the record. The following record, JPI_LIST, addresses a single field of 12 bytes.



At a second level, the names BUFLEN, CODE, BUFADR, and RETLENADR may be used to address individual fields in the JPI_LIST record.



To use a record variable, take the following steps:

1. Define a record structure using a structure block (see Section 6.10.1).
2. Declare a record variable using a RECORD statement (see Section 6.10.2).
3. Reference the record by using the record variable name, or a field of the record by using the record variable name followed by a period and a record field name (see Section 6.10.3).

The following program segment uses a record to pass an item list to the SYS$GETJPIW system service. The definition portion of the program defines a record structure ITMLST and a record variable JPI_LIST of type ITMLST. The ITMLST structure alternately contains four fields (BUFLEN, CODE, BUFADR, and RETLENADR) or one field (END_LIST); the previous figure shows the four-field alternative. The execution portion of the program assigns values to each field of the JPI_LIST record and then passes that record to the SYS$GETJPIW system service.

```
! Structure definition
IMPLICIT NONE
INCLUDE '($JPIDEF)'
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Record declaration
RECORD /ITMLST/ JPI_LIST (2)
! Buffers for SYS$GETJPI
INTEGER*4 PRIORITY
INTEGER*4 PRIORITY_LEN
INTEGER*4 STATUS
INTEGER*4 SYS$GETJPIW
! Assign field values
JPI_LIST(1).BUFLEN = 4
JPI_LIST(1).CODE = JPI$_PRI
JPI_LIST(1).BUFADR = %LOC(PRIORITY)
JPI_LIST(1).RETLENADR = %LOC(PRIORITY_LEN)
JPI_LIST(1).END_LIST = 0
! Call SYS$GETJPIW
STATUS = SYS$GETJPIW (,,,
2                      JPI_LIST,,,)
```

FORTRAN stores a record in memory as a sequence of values. The first field of the record is in the first storage location and the last field in the last storage location. No gaps are left between storage locations. In an array of records, each element in the array is stored in a similar fashion with no gaps between array elements.

6.10.1 Defining Record Formats

A record's structure is defined by the number, order, and data types of the fields within the record. To define a record structure, use a structure block of the following form:

```
STRUCTURE /structure-name/
  field-declaration
  [field-declaration]
            .
            .
            .
END STRUCTURE
```

The STRUCTURE statement begins a structure block and names the record structure. Each **field-declaration** defines one field of the structure; the order of the declarations determines the order of the fields. Field names must be unique within a structure block. The END STRUCTURE statement terminates the structure block. A structure block does not create any variables; it defines a record format and names the fields within that record format. To create a record variable, use a RECORD statement as shown in Section 6.10.2.

Note

Since periods are used in record references to separate fields, you should avoid using the names of relational operators (.EQ., .XOR., etc.), logical constants (.TRUE. or .FALSE.), or logical operators (.AND., .NOT., etc.) as field names.

A record field can contain typed data, a substructure, or a union. Field types can vary from field to field within a record. The three different field types are described in the following subsections.

6.10.1.1 Typed Data

A variable or an array of any FORTRAN data type. **Field-declarations** for typed data take the form of data type declaration statements. Adjustable or assumed-size arrays and passed-length character strings are illegal within a structure block. The following example defines a record structure containing three fields: the first is a BYTE field named CLASS, the second an INTEGER*2 field named WIDTH, and the third a BYTE array field named LENGTH.

```
STRUCTURE /CHARACTERISTICS/
  BYTE      CLASS
  INTEGER*2 WIDTH
  BYTE      LENGTH(4)
END STRUCTURE
```

Any required array dimensions must be specified in the field declaration statements; DIMENSION statements cannot be used to define field names.

The IMPLICIT statement has no effect on field names; you must explicitly name each field in a **field-declaration**. By default, field values are initially undefined. To specify an initial field value, enclose a value in slashes following the field name, as shown in Section 2.1.2. (DATA statements do not permit record references.) If a field value is initialized in a structure block, all records declared using that structure receive the initial field values.

FORTRAN provides one pseudofield name, %FILL, which creates an empty space in the record. The following example uses %FILL to create a one-byte space between the FLAG and WORD fields.

```
STRUCTURE /ALIGNED/
  BYTE      FLAG,
2           %FILL
  INTEGER*2 WORD
END STRUCTURE
```

6.10.1.2 Substructures

A record. If you specify a record structure as a field value within another record, the inner record structure (substructure) cannot have the same structure name as that of the outer record structure (that is, a structure definition cannot include itself at any level of nesting). **Field-declarations** for substructures can be either RECORD statements or structure blocks.

If the **field-declaration** is a RECORD statement, the specified structure name must have been previously defined. If the **field-declaration** is a structure block, the field names (with the exception of %FILL) must be unique within the substructure.

The following example defines a structure APPOINTMENT that contains two fields: the first field, APP_DATE, is a substructure of type DATE; the second field, APP_TIME, is a substructure of type TIME. Note that a STRUCTURE statement within another structure block can include a list of field names; the substructure block defines the format of each field in the list.

```
STRUCTURE /DATE/
 INTEGER*4 DAY,
2          MONTH,
2          YEAR
END STRUCTURE
STRUCTURE /APPOINTMENT/
 RECORD /DATE/ APP_DATE      ! Field one
 STRUCTURE /TIME/ APP_TIME   ! Field two
  INTEGER*4 HOUR,
2           MINUTE
 END STRUCTURE
END STRUCTURE
```

6.10.1.3 Unions

A union is two or more fields logically sharing a common location. **Field-declarations** for unions take the following form:

```
STRUCTURE /structure-name/
 UNION
  map-declaration
  [map-declaration]
          .
          .
          .
 END UNION
END STRUCTURE
```

Map-declarations take the following form:

```
MAP
 field-declaration
 [field-declaration]
          .
          .
          .
END MAP
```

Multiple maps share the same area of the record structure. The size of the shared area is the size of the largest map in the UNION block. At any time during program execution, the fields of exactly one map within a union are defined. However, if you overlay one map with a smaller map, any part of the larger map that is not overlaid remains unchanged.

The following example sets up a record to receive terminal characteristics. Since the basic characteristics are returned in the low-order three bytes of the fourth field (BASIC field) and the length of the record is returned in the high-order byte of the same field (fourth element of the LENGTH field), a union acts as an equivalence for the BASIC and LENGTH fields.

```
STRUCTURE /CHARACTERISTICS/
  BYTE      CLASS,     ! Field 1
2           TYPE       ! Field 2
  INTEGER*2 WIDTH      ! Field 3
  UNION                ! Field 4
   MAP
    INTEGER*4 BASIC
   END MAP
   MAP
    BYTE LENGTH(4)
   END MAP
  END UNION
  INTEGER*4 EXTENDED   ! Field 5
END STRUCTURE
```

The effect of a union is different from that of an EQUIVALENCE statement in that the data entities in an EQUIVALENCE statement are concurrently associated with a common storage location whereas the maps in a union block are alternately associated with a common storage location.

Another common use of a union block is to provide one or more different definitions for the same record. For example, many system services require that you pass information in an item list consisting of one or more items (each containing four separate fields) followed by a longword containing a value of 0 to indicate the end of the item list. In order to specify an item list as an array of records, you must be able to use the same record for the four-element items and the one-element end of list. In the following structure, a union provides the required double definition.

```
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST /0/
  END MAP
 END UNION
END STRUCTURE
```

6.10.2 Declaring Record Variables

To declare a record variable, use a RECORD statement of the following form:

```
RECORD /structure-name/ record-name-list
2     [,/structure-name/ record-name-list]
                          .
                          .
                          .
```

Structure-name must be the name of a record structure previously defined in a structure block (see Section 6.10.1). **Record-name-list** is a list of one or more variable names, array names, or array declarators, separated by commas (adjustable arrays are permitted, provided that the array is a dummy argument). The following statements create three record items, a record variable DVI_LIST and four-element record array JPI_LIST (both of type ITMLST), and a record variable IOSTAT (type IOSB).

```
! Item list for SYS$GETDVI and SYS$GETJPI
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
RECORD /ITMLST/ DVI_LIST,    ! Item 1
2               JPI_LIST(4)  ! Item 2
! I/O status block
STRUCTURE /IOSB/
 INTEGER*2 STATUS
 INTEGER*2 NOTHING
 INTEGER*4 ZERO /0/
END STRUCTURE
RECORD /IOSB/ IOSTAT         ! Item 3
```

6.10.3 Referencing Records

Records can be referenced in one of two ways:

- Qualified reference—Refers to a typed record field; that is, a field that is not itself a record.

  `record-name[.subrecord-name...].typed-name`

- Unqualified reference—Refers to a record structure or record substructure.

  `record-name[.subrecord-name...]`

Typically, you can use a qualified record reference in any context that allows a variable of the same data type. However, you cannot use a qualified record reference in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements.

The contexts in which you can use unqualified record references are more limited.

- Assignment statements—Specifies that each field of the record on the right-hand side of the assignment statement is to be assigned to the matching field of the record on the left-hand side. The value on either side of the assignment statement must be a record; both records must have been declared using the same structure name.
- COMMON—Specifies that the record is to be placed in the common block.

- FUNCTION—You can specify a record as either a dummy or an actual argument. The number, type, and order of the fields in a dummy record argument must match those of the actual argument (the order of maps within a union does not matter).
- SUBROUTINE—You can specify a record as either a dummy or an actual argument. The number, type, and order of the fields in a dummy record argument must match those of the actual argument (the order of maps within a union does not matter).
- ENTRY—You can specify a record as either a dummy or an actual argument. The number, type, and order of the fields in a dummy record argument must match those of the actual argument (the order of maps within a union does not matter).
- VOLATILE—Prevents FORTRAN from performing optimization operations on the record.
- I/O—You can specify unqualified record references in unformatted I/O statements, but not in formatted, list-directed, or NAMELIST I/O statements.

When mixing record and array notation, take care where you place the subscript. If you want to reference one element of an array of records, place the subscript after the record name but before any field names.

```
STRUCTURE /ORDER_REC/
  BYTE         FLAGS(3)
  CHARACTER*80 CUSTOMER
  INTEGER*4    ITEM_CODE
END STRUCTURE
RECORD /ORDER_REC/ ORDERS(2500)

ORDERS(1).ITEM_CODE
```

If you want to reference one element of an array that is a field within a record, place the subscript after the field name.

```
STRUCTURE /ORDER_REC/
  BYTE         FLAGS(3)
  CHARACTER*80 CUSTOMER
  INTEGER*4    ITEM_CODE
END STRUCTURE
RECORD /ORDER_REC/ TEMPORARY

TEMPORARY.FLAGS(1)
```

6.10.4 Storing Record Structures

If you have a number of commonly used record structures, you can store the structures in a text library and use the FORTRAN INCLUDE statement to reference them from your programs when necessary. One drawback to specifying record definitions with the INCLUDE statement is that a person reading your program cannot examine the field definitions of the record. A common candidate for inclusion in a structure library would be the item list structure used by many of the system services, as shown here.

ITMLST.TXT

```
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            RETLENADR
  END MAP
 END UNION
END STRUCTURE
```

Create a text library for your structures and put the ITMLST.TXT file in the library.

```
$ LIBRARY/CREATE/TEXT $DISK1:[DEV.LIBRARY]STRUCTURES
$ LIBRARY/REPLACE/TEXT $DISK1:[DEV.LIBRARY]STRUCTURES ITMLST
```

To include the ITMLST structure block in a program, use the INCLUDE statement as shown in the following example. Note that you still need the RECORD statement to define a record variable of type ITMLST.

```
! Include the ITMLST structure definition
INCLUDE 'DISK1:[DEV.LIBRARY]STRUCTURES (ITMLST)'
! Record declaration
RECORD /ITMLST/ JPI_LIST
! Buffers for SYS$GETJPI
INTEGER*4 PRIORITY,
2         PRIORITY_LEN

! Assign field values
JPI_LIST.BUFLEN = 4
JPI_LIST.CODE = JPI$_PRI
JPI_LIST.BUFADR = %LOC(PRIORITY)
JPI_LIST.RETLENADR = %LOC(PRIORITY_LEN)
! Call SYS$GETJPIW
STATUS = SYS$GETJPIW (,,,
2                     JPI_LIST,,,)
```

6.11 System Information

The VAX/VMS operating system provides services for collecting information from the system.

6.11.1 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 the statistics by invoking the following Run-Time Library procedures:

- LIB$INIT_TIMER—Allocates and initializes space for collecting the statistics. You should specify the one argument and specify it as an integer variable with a value of 0 to ensure the modularity of your program (for example, in case a program unit calling your program unit also sets a timer). When you specify the argument, the system collects the information in a specially allocated area in dynamic storage.
- LIB$SHOW_TIMER—Obtains one statistic or all the statistics; the statistics are formatted for output. The first argument must be the same integer variable you specified for LIB$INIT_TIMER (do not modify this variable). Specify the second argument to obtain one particular statistic rather than all the statistics.

  You can let the system write the statistics to SYS$OUTPUT (the default) or you can process the statistics with a subprogram of your own. To process the statistics yourself, specify the name of your subprogram as the third argument (make sure you declare it as EXTERNAL and INTEGER*4). You can pass one argument to your subprogram by naming it as the fourth argument to LIB$SHOW_TIMER. 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$STAT_TIMER—Obtains one unformatted statistic. Specify the statistic as the first argument. Specify a storage area for the statistic as the second argument: an array of two INTEGER*4 data items for the elapsed time and one INTEGER*4 data item for each of the other statistics. The last argument must be the same integer variable you specified for LIB$INIT_TIMER.
- LIB$FREE_TIMER—You should invoke this procedure when you are done with the timer to ensure the modularity of your program. The argument must be the same integer variable specified for LIB$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. The following example invokes LIB$SHOW_TIMER and uses a user-written subprogram to either display the statistics or write them to a file.

```
! 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
```

```
! 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 services SYS$GETTIM and SYS$GETSYI to obtain more detailed system information.

6.11.2 System Time

The VAX/VMS operating system recognizes two types of time:

- Absolute time—A certain date and/or time of day.
- Delta time—A number of days and/or units of time within a day.

The VAX/VMS operating system formats an absolute time as follows. 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
ss.ss	Second and hundredth of second

The VAX/VMS operating system formats a delta time as follows. The full date and time require a character string of 16 characters. The punctuation is required.

```
dddd hh:mm:ss.ss
```

dddd	Days
hh	Hours
mm	Minutes
ss.ss	Seconds and hundredths of seconds

Internally the system maintains an 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. (In FORTRAN, define an internal time as an array of two INTEGER*4 variables.)

6.11.2.1 Current Time

The Run-Time Library procedure LIB$DATE_TIME returns a character string containing the current date and time in absolute time format. The full string requires a declaration of CHARACTER*23. If you specify a shorter string, the value is truncated. A declaration of CHARACTER*16 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$FAO system service, and back to internal format with the SYS$BINTIM system service.

6.11.2.2 Time Manipulation

The 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) are as follows:

- 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.
- Manipulate the times—Add, subtract, or otherwise manipulate the times. Use the Run-Time Library procedures LIB$ADDX and LIB$SUBX to add and subtract times, since the times are defined in integer arrays. 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.
- Format the times—Format the result, as desired, with SYS$BINTIM or SYS$FAO.

The following example 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 procedure. For the absolute or delta time format, see Section 6.11.2.

```
! 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 the date/time (that is, working just at the date level), the Run-Time Library procedure LIB$DAY might simplify your calculations. LIB$DAY returns to you the number of days from the base system date to a given date.

6.11.3 Emulated Instructions

A number of VAX machine code instructions are implemented by special hardware. If your machine does not have the hardware, the missing instructions are emulated by the system software (at slower speeds). The MicroVMS systems emulate D_floating, H_floating, decimal, and string instructions.

If execution speed is important to your program, you may wish to execute code conditionally depending on whether the machine executing the program implements a certain set of instructions using hardware or software. The LIB$GETSYI and SYS$GETSYI system-defined procedures allow you to use the following item codes to determine which types of instructions are emulated.

Item code	Data type
SYI$_D_FLOAT_EMULATED	D_floating instructions
SYI$_F_FLOAT_EMULATED	F_floating instructions
SYI$_G_FLOAT_EMULATED	G_floating instructions
SYI$_H_FLOAT_EMULATED	H_floating instructions
SYI$_DECIMAL_EMULATED	Decimal instructions
SYI$_CHARACTER_EMULATED	Character instructions

The following program segment uses LIB$SETSYI to check on D_floating instructions. If D_floating instructions are emulated, LIB$GETSYI returns a logically true (odd) value in the INTEGER variable EMULATE.

```
INTEGER STATUS,
2       LIB$GETSYI,
2       EMULATE
EXTERNAL SYI$_D_FLOAT_EMULATED
STATUS = LIB$GETSYI (%LOC(SYI$_D_FLOAT_EMULATED),
2                    EMULATE)
IF (EMULATE) THEN
           .
           .
           .
END IF
END
```

The F$GETSYI lexical function returns the same information at DCL command level.

```
$ WRITE SYS$OUTPUT F$GETSYI ("D_FLOAT_EMULATED")
FALSE
```

7 Command Input and Syntax Analysis

To invoke a program using a unique DCL command, define the command in a command language definition (CLD) file, then apply the CLD file to a command table with the SET COMMAND command. For details about CLD source statements and the SET COMMAND command, see the description of the Command Definition Utility in the *VAX/VMS Command Definition Reference Manual.*

In your program, use the Command Language (CLI) routines to examine the command line typed by the user to invoke your program. For details about the CLI routines, see the *VAX/VMS Utility Routines Reference Manual.*

7.1 Command Description

Define a command by writing a CLD file. The default file type for a CLD file is CLD. You can define more than one command in a file.)

7.1.1 Command Name and Image

Start a command definition with a DEFINE VERB statement followed optionally by an IMAGE clause. You can supply alternate names for the command with SYNONYM clauses. Ensure that the command name and any synonyms you choose are unique to four characters among all commands in your system. The following example defines a command named INCOME whose image (the executable program to be invoked when INCOME is entered as a DCL command) is in the file WORKDISK:[INCOME]INCOME.EXE (the file type of the image defaults to EXE).

```
DEFINE VERB INCOME
IMAGE "WORKDISK:[INCOME]INCOME"
```

If you do not enter an IMAGE statement, the image file name defaults to the command name, the directory to SYS$SYSTEM, and the file type to EXE. For example, if the IMAGE statement were omitted from the previous definition, the image would have to exist as the file SYS$SYSTEM:INCOME.EXE. When an IMAGE statement is omitted or specifies only a file name, you can use a logical name identical to the command or the specified file name to point to the image; that is, the equivalence name is assumed to be the image file. For example, if the IMAGE statement were omitted from the above definition, the user could invoke the proper image by defining the following logical name prior to invoking the command.

```
$ DEFINE INCOME WORKDISK:[INCOME]INCOME
```

Specifying the full file specification of the image in the CLD file precludes logical name accidents. For example, suppose you omit the IMAGE statement for INCOME and put the image file in SYS$SYSTEM. If the user defines INCOME for some other purpose, the command will not work. Not specifying the name of the image in the CLD file does permit you to relocate the image or use different versions of the image without redefining the command.

7.1.2 Parameters

Specify a parameter with the PARAMETER clause. Parameters must be named P1, P2, and so on, to a maximum of 8. You must not omit parameter 1 or skip a parameter although you can specify parameters out of order. The following example indicates that the INCOME command takes two parameters.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
PARAMETER P2
```

7.1.2.1 Labels

The LABEL clause provides a means for your program to refer to the parameter. The label defaults (if LABEL is not specified) to the parameter name (P1, P2, and so on). The following example provides labels other than P1 and P2 for the parameters.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE
PARAMETER P2
          LABEL = REPORT_FILE
```

7.1.2.2 Required Parameters

You can make a parameter required by specifying the REQUIRED option of the VALUE clause. All required parameters must be specified before any optional parameters. If an interactive user omits a required parameter, DCL enters prompting mode and prompts for the required parameter. (Section 7.1.2.4 describes prompt mode.) To continue command execution, the user must enter a value. (The user can enter two consecutive quotation marks ("") to indicate a null value.) If a noninteractive user omits a required parameter, an error occurs.

The following example makes the two INCOME parameters required. You can specify the prompt with the PROMPT clause, as shown in the example, or let the prompt default to the parameter label. In either case, the prompt appears on the terminal preceded by an underscore and followed by a colon and a tab.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE
          VALUE (REQUIRED)
          PROMPT = "Statistics file"
PARAMETER P2
          LABEL = REPORT_FILE
          VALUE (REQUIRED)
          PROMPT = "Report file"
```

7.1.2.3 Values

If you do not specify the REQUIRED option, the parameter is optional. You can specify a default value for an optional parameter with the DEFAULT option of the VALUE clause. (Defaulting keyword values is different; see Section 7.1.4.2.) To permit entry of a list of values separated by commas or plus signs specify the LIST option of the VALUE clause.

The following example makes the two INCOME parameters optional and provides default values for them. The example permits the user to enter a list of values for P2. You can provide one default value for a list.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE
          VALUE (DEFAULT = "STATS.SAV")
PARAMETER P2
          LABEL = REPORT_FILE
          VALUE (LIST, DEFAULT = "INCOME.RPT")
```

Specify CONCATENATE instead of LIST to permit entry of values separated only by plus signs. Specify LIST and NOCONCATENATE to permit entry of values separated by commas only.

7.1.2.4 Prompting

When an interactive user omits a required parameter, DCL enters prompt mode and prompts for the required parameter. (Omitting a required parameter while working noninteractively causes an error.) A user must respond by pressing RETURN to repeat the prompt, pressing CTRL/Z to abort the command, or entering a value. If the user enters a legal value, DCL prompts for the next parameter, if any. If this parameter is required, the user must respond as described. If the parameter is optional, pressing RETURN executes the command; pressing CTRL/Z aborts the command; and entering a legal value causes DCL to prompt for the next parameter, if any.

Use the PROMPT clause to specify a prompt for a required or optional parameter, or let the prompt default to the parameter label. Optional parameter prompts are displayed only if a required parameter has been omitted, causing DCL to enter prompt mode.

7.1.3 Qualifiers

Specify a qualifier with the QUALIFIER clause. The name of the qualifier immediately follows the word QUALIFIER; do not put a slash at the beginning of the name. The following example indicates that the income command takes three qualifiers (and one parameter).

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE
          VALUE (DEFAULT = "STATS.SAV")
QUALIFIER ENTER
QUALIFIER FIX
QUALIFIER REPORT
```

The name of the qualifier serves as the label unless you explicitly provide an alternative name with the LABEL clause.

7.1.3.1 Negatable Qualifiers

By default, a qualifier is negatable: a user can prefix the qualifier name with NO to explicitly negate its presence in the command line. You can make a qualifier nonnegatable with the NONNEGATABLE option. The following example makes the INCOME qualifiers nonnegatable.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE,
          VALUE (DEFAULT = "STATS.SAV")
QUALIFIER ENTER
          NONNEGATABLE
QUALIFIER FIX
          NONNEGATABLE
QUALIFIER REPORT
          NONNEGATABLE
```

7.1.3.2 Default Presence

A qualifier is present by default if you specify the DEFAULT option. A qualifier is present by default only in batch jobs if you specify the BATCH option. The user must explicitly negate a default qualifier to make it not present. (However, your program can detect whether a qualifier is entered explicitly or is present by default.) The following example makes the ENTER qualifier present by default.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE,
          VALUE (DEFAULT = "STATS.SAV")
QUALIFIER ENTER
          DEFAULT
QUALIFIER FIX
QUALIFIER REPORT
```

7.1.3.3 Placement

You can restrict the placement of a qualifier in the command line by specifying one of the following PLACEMENT options:

- GLOBAL (default)—The qualifier affects all parameters no matter where the user places it on the command line.
- LOCAL—The qualifier can only be placed after a parameter. The qualifier can be placed after more than one parameter. The qualifier affects only the parameter it follows.
- POSITIONAL—The qualifier can be placed before all parameters or after one or more parameters. The qualifier affects all parameters if it precedes all parameters. Otherwise, the qualifier affects only the parameter it follows.

The following example permits varying interpretations of the qualifier depending on whether it is positioned before all parameters or follows a particular parameter.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1, VALUE (REQUIRED)
              PROMPT = "Input file"
PARAMETER P2, VALUE (REQUIRED)
              PROMPT = "Output file"
QUALIFIER FILETYPE, PLACEMENT = POSITIONAL
```

See Section 7.3.3 for the programming techniques involved in detecting the position of a local or positional qualifier.

7.1.3.4 Values

The VALUE clause permits the user to specify a value for the qualifier (/qualifier-name = value). The REQUIRED option of the VALUE clause makes specification of the value required. If the value is optional (REQUIRED is not specified), you can specify a default value with the DEFAULT option. (Defaulting keyword values is different; see Section 7.1.4.2). The LIST option permits entry of a list of values (/qualifier = (value,...)).

The following example permits the user to enter a list of values on the REPORT qualifier; you can provide one default value for a list.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE,
          VALUE (DEFAULT = "STATS.SAV")
QUALIFIER ENTER
QUALIFIER FIX
QUALIFIER REPORT
          VALUE (LIST, DEFAULT = "INCOME.RPT")
```

7.1.4 Value Types

You can require that a parameter, qualifier, or keyword value conform to the specifications of a built-in value type, or that a value be one of a list of keywords.

7.1.4.1 Built-in Value Types

Identify the built-in value types with the system-defined names: $ACL, $DATETIME, $DELTATIME, $FILE, $NUMBER, $QUOTED_STRING, and $REST_OF_LINE. For details, see the description of the Command Definition Utility in the *VAX/VMS Command Definition Reference Manual*. The following example requires the parameter value to be a valid file specification.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILE
          VALUE (TYPE = $FILE,
                 DEFAULT = "STATS.SAV")
```

7.1.4.2 Keywords

Define a list of keywords by creating a DEFINE TYPE block consisting of a DEFINE TYPE statement followed by KEYWORD statements. To indicate that the user must enter only listed keywords (or unique abbreviations of the keywords), use the TYPE option of the VALUE clause to specify the name of the data type described by the DEFINE TYPE statement. The following example gives the user a choice of entering ENTER, FIX, or REPORT for the parameter value.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = ACTION
          VALUE (TYPE = ACTION_TYPE)
DEFINE TYPE ACTION_TYPE
            KEYWORD ENTER
            KEYWORD FIX
            KEYWORD REPORT
```

Keywords can be modified by most of the same clauses as qualifiers: DEFAULT, LABEL, NEGATABLE, NONNEGATABLE, SYNTAX, and VALUE. However, NONNEGATABLE is the default. The VALUE clause can take the DEFAULT, LIST, REQUIRED, and TYPE options.

To specify a default keyword:

1 In the PARAMETER or QUALIFIER statement, use a DEFAULT clause to indicate that a default value exists.

2 In the DEFINE TYPE block, use a DEFAULT clause in the appropriate KEYWORD statement to identify the default keyword.

The following example requires that the user enter ALL or FILE = (filename,...) for the parameter value. The default keyword is ALL. The default value of the FILE keyword is STATS.SAV.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
PARAMETER P1
          LABEL = STATS_FILES
          VALUE (TYPE = FILE_TYPE)
          DEFAULT
DEFINE TYPE FILE_TYPE
KEYWORD ALL
        DEFAULT
KEYWORD FILE
        VALUE (TYPE = $FILE, LIST, DEFAULT = "STATS.SAV")
```

If you are defaulting a keyword value for a qualifier or keyword that you do not want defaulted, omit step one. Since a parameter value cannot be defaulted without being present, you cannot omit step one when defaulting a keyword value for a parameter.

7.1.5 Disallowing Entities and Combinations

You can prohibit specified parameters, qualifiers, and keywords by naming the restricted entities in DISALLOW statements. You can prohibit certain combinations of entities by using expressions in DISALLOW statements. The following example permits only one of the three qualifiers to be entered.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
QUALIFIER ENTER, NONNEGATABLE
QUALIFIER FIX, NONNEGATABLE
QUALIFIER REPORT, NONNEGATABLE
DISALLOW ANY2 (ENTER, FIX, REPORT)
```

For a complete list of expressions you can use in DISALLOW statements, see the description of the Command Definition Utility in the *VAX/VMS Command Definition Reference Manual.*

7.1.6 Syntax Changes

You can change the syntax requirements of a command depending on the entry of a particular parameter, qualifier, or keyword. The parameter, qualifier, or keyword points to a new syntax described in a DEFINE SYNTAX block. The following example invokes one of three different images depending on the value entered for P1. The /OUTPUT qualifier is valid only if REPORT is entered as the action.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:ENTER"
NOQUALIFIERS
PARAMETER P1
          LABEL = ACTION
          VALUE (TYPE = ACTION_TYPE)
          DEFAULT
DEFINE TYPE ACTION_TYPE
KEYWORD ENTER, DEFAULT
KEYWORD FIX, SYNTAX = FIX_SYNTAX
KEYWORD REPORT, SYNTAX = REPORT_SYNTAX
DEFINE SYNTAX FIX_SYNTAX
IMAGE "SYS$SYSTEM:FIX"
DEFINE SYNTAX REPORT_SYNTAX
IMAGE "SYS$SYSTEM:REPORT"
QUALIFIER OUTPUT
          VALUE (TYPE = $FILE)
```

If you specify new parameter definitions, those definitions replace all the old parameter definitions. If you specify NOPARAMETERS, the new syntax permits no parameters. If you specify no new parameters and do not specify NOPARAMETERS, the old parameter definitions remain in effect.

If you specify new qualifier definitions, those definitions replace all the old qualifier definitions. If you specify NOQUALIFIERS, the new syntax permits no qualifiers. If you specify no new qualifiers and do not specify NOQUALIFIERS, the old qualifier definitions remain in effect.

If you specify new DISALLOW definitions, those definitions replace all the old DISALLOW definitions. If you specify NODISALLOW, the new syntax permits no disallows. If you specify no new disallow definitions and do not specify NODISALLOW, the old DISALLOW definitions remain in effect.

7.1.7 Keyword and Definition Paths

Keyword and definition paths permit you to identify entities exactly.

7.1.7.1 Keyword Paths

To refer to keywords that would otherwise be ambiguous, construct keyword paths. A keyword path is constructed by preceding the ambiguous entity with the name of the parent parameter, qualifier, or keyword and a period. A path can be constructed to a depth of 8. The following example uses keyword paths to distinguish the keywords for the /INPUT qualifier from those for the /OUTPUT qualifier. Entering INCOME/INPUT=FORM1 /OUTPUT=FORM3, for example, is illegal.

```
DEFINE VERB INCOME
IMAGE "SYS$SYSTEM:INCOME"
QUALIFIER INPUT, VALUE (TYPE = FILE_TYPE)
QUALIFIER OUTPUT, VALUE (TYPE = FILE_TYPE)
DISALLOW INPUT.FORM1 AND OUTPUT.FORM3
DISALLOW INPUT.FORM2 AND OUTPUT.FORM3
DEFINE TYPE FILE_TYPE
KEYWORD FORM1
KEYWORD FORM2
KEYWORD FORM3
```

7.1.7.2 Definition Paths

To refer to parameter values, qualifiers, or keywords not defined in the same DEFINE VERB or DEFINE SYNTAX block, construct definition paths. In a definition path, you precede the entity name by the name of the parent verb or syntax in angle brackets. Definition paths are useful for referring to the qualifiers in an old syntax from a new syntax (as long as the qualifiers are not disallowed by the specification of new qualifiers or the NOQUALIFIERS characteristic). The following example disallows certain inherited qualifiers in the new syntaxes. The user cannot enter INCOME/ENTER/OUTPUT, for example.

```
DEFINE VERB INCOME
QUALIFIER ENTER, NONNEGATABLE, SYNTAX = ENTER_SYNTAX
QUALIFIER FIX, NONNEGATABLE, SYNTAX = FIX_SYNTAX
QUALIFIER REPORT, NONNEGATABLE, SYNTAX = REPORT_SYNTAX
DISALLOW ANY2 (ENTER, FIX, REPORT)
QUALIFIER INPUT, VALUE (TYPE = FILE_TYPE)
QUALIFIER OUTPUT, VALUE (TYPE = FILE_TYPE)
DEFINE TYPE FILE_TYPE
KEYWORD FORM1
KEYWORD FORM2
KEYWORD FORM3
DEFINE SYNTAX ENTER_SYNTAX
IMAGE "SYS$SYSTEM:ENTER"
DISALLOW <INCOME>OUTPUT
DEFINE SYNTAX FIX_SYNTAX
IMAGE "SYS$SYSTEM:FIX"
DISALLOW <INCOME>INPUT OR <INCOME>OUTPUT
DEFINE SYNTAX REPORT_SYNTAX
IMAGE "SYS$SYSTEM:REPORT"
DISALLOW <INCOME>INPUT
```

7.2 Command Setup

The commands which a user can type at DCL command level are defined by the user's process command table. The table exists only for the duration of the process and is recreated each time the user logs in or otherwise creates a process. By default, the process command table is created from the file SYS$LIBRARY:DCLTABLES.EXE (the DCL command table) when the process is created.

7.2.1 Process Command Table

To add or replace commands in your current process command table, specify the name of a CLD file in the SET COMMAND command. Each command defined in the file by a DEFINE VERB statement is added to the table if the command does not already exist, or is substituted in the table if the command already exists. The following example adds or replaces the INCOME command (defined in the file INCOME.CLD) in the process command table.

```
$ SET COMMAND WORKDISK:[INCOME]INCOME
```

Adding or replacing a command in the process command table makes the command available until the process terminates. (To make the command available each time you log in, place the SET COMMAND command in your login command file.)

7.2.2 DCL Command Table

To add or replace a command in the DCL command table, use the SET COMMAND command specifying the DCL table as input with the /TABLE qualifier and also specifying the DCL table as output with the /OUTPUT qualifier. You must have WRITE access to DCLTABLES.EXE in SYS$LIBRARY. The following example adds or replaces the INCOME command in the DCL command table.

```
$ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES-
_$ /OUTPUT=SYS$LIBRARY:DCLTABLES -
_$ WORKDISK:[INCOME]INCOME
```

Modifications to the DCL command table affect all users on the system (except for users who switch to a user command table; see below). For each user, the changes become effective the next time the user logs in. A user can make the changes effective immediately by entering the following command:

```
$ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES
```

7.2.3 User Command Table

You can create and use your own command tables.

7.2.3.1 Copying the DCL Command Table

To create a copy of the DCL command table, either use the COPY command or use the SET COMMAND command specifying the DCL command table as input with the /TABLE qualifier and the name of a file to hold the copy as output with the /OUTPUT qualifier. The following command creates a copy of the DCL table SYS$LIBRARY:DCLTABLES.EXE in the [INCOME] directory on WORKDISK.

```
$ SET COMMAND/TABLE = SYS$LIBRARY:DCLTABLES-
_$ /OUTPUT = WORKDISK:[INCOME]DCLTABLES
```

To add a command or commands to your copy of the table while you are copying it, specify the name of the CLD file containing the command in the SET COMMAND. The following example creates a copy of the DCL table and adds the INCOME command to the new table.

```
$ SET COMMAND/TABLE = SYS$LIBRARY:DCLTABLES-
_$ /OUTPUT = WORKDISK:[INCOME]DCLTABLES -
_$ WORKDISK:[INCOME]INCOME
```

To add or replace commands in the copied table after you have copied it, use the SET COMMAND specifying the file name of the copied table as input with the /TABLE qualifier and as output with the /OUTPUT qualifier. The following example adds the INCOME command to a copy of the DCL command table.

```
$ SET COMMAND/TABLE = WORKDISK:[INCOME]DCLTABLES-
_$ /OUTPUT = WORKDISK:[INCOME]DCLTABLES -
_$ WORKDISK:[INCOME]INCOME
```

7.2.3.2 Establishing the User Table

To establish your copy of the DCL command table as the process command table rather than DCLTABLES.EXE in SYS$LIBRARY, use the SET COMMAND command specifying your copy of the table as input with the /TABLE qualifier. Do not specify a parameter and do not specify the /OUTPUT qualifier. The following example switches your process command table to a user table.

```
$ SET COMMAND/TABLE = WORKDISK:[INCOME]DCLTABLES
```

You must switch tables each time you log in except in the following cases. If the user table is in the system directory SYS$LIBRARY, you can specify that it be used as the process table at login instead of the default DCL table with the /CLITABLES qualifier to AUTHORIZE. You can also specify an alternative table (again the table must be in SYS$LIBRARY) by following your user name with the /TABLES qualifier when you log in.

7.2.3.3 Creating a Table from Scratch

Using a table that is not a copy of the DCL table makes the VAX/VMS commands unavailable and is therefore not recommended. If you do create your own command table and wish to reestablish the DCL command table, you must log out and log back in again. Since LOGOUT is a VAX/VMS command, you will have to delete your process by using the STOP command from another terminal or, if your system has only one terminal, by bringing down the system.

To create your own command table from scratch, create an empty table, add the commands that you want in the table, and then switch to your table as outlined in Section 7.2.3.3. To create an empty table, create a CLD file that contains a MODULE statement and optionally an IDENT statement; compile the CLD file with a SET COMMAND/OBJECT command; and link the resultant object module as a shareable image with the LINK/SHARE command. The following commands create an empty command table.

```
$ SET DEFAULT WORKDISK:[INCOME]
$ CREATE INCTABLES.CLD
 MODULE INCOME_TABLES <ctrl/z>
$ SET COMMAND/OBJECT INCTABLES
$ LINK/SHARE INCTABLES
```

Add the desired commands to your newly created table. The following example adds the INCOME command.

```
$ SET COMMAND/TABLE = WORKDISK:[INCOME]INCTABLES-
_$ /OUTPUT = WORKDISK:[INCOME]INCTABLES -
_$ WORKDISK:[INCOME]INCOME
```

You are now ready to switch to your table as outlined in Section 7.2.3.3. Remember, once you switch to your table, no VAX/VMS commands are available.

7.2.4 Deleting Commands

Use the /DELETE qualifier of the SET COMMAND command to delete a command from a table. Specify the /TABLE and /OUTPUT qualifiers as necessary to work on tables in files rather than your process command table. The following example deletes the INCOME command from your process table.

```
$ SET COMMAND/DELETE=INCOME
```

The next example deletes the INCOME command from the DCL table.

```
$ SET COMMAND/TABLE=SYS$LIBRARY:DCLTABLES-
_$ /OUTPUT=SYS$LIBRARY:DCLTABLES-
_$ /DELETE=INCOME
```

7.3 Syntax Analysis

The Run-Time Library routines CLI$PRESENT and CLI$GET_VALUE permit you to check for the presence of parameters, qualifiers, and keywords, and to read the values typed by the user for those elements.

7.3.1 Checking for the Presence of Elements

CLI$PRESENT takes one argument: a character string whose value is the label of a parameter, qualifier, or keyword as specified in the CLD file. CLI$PRESENT returns a status of success if the user entered the element or if the element is present by default. CLI$PRESENT returns a status of failure (warning) if the user did not enter the element (and the element is not present by default) or if the user explicitly negated the element. The following (an excerpt from a CLD file) example performs different actions depending on which of three qualifiers the user enters.

Excerpt from CLD File

```
QUALIFIER ENTER
QUALIFIER FIX
QUALIFIER REPORT
DISALLOW ANY2 (ENTER, FIX, REPORT)
```

Excerpt from Program

```
INTEGER CLI$PRESENT
        .
        .
        .
IF (CLI$PRESENT ('ENTER')) THEN
        .
        .
        .
ELSE IF (CLI$PRESENT ('FIX')) THEN
        .
        .
        .
ELSE IF (CLI$PRESENT ('REPORT')) THEN
        .
        .
        .
END IF
```

If the distinction between explicit entry and default presence of an element, or explicit negation and omission of an element, is significant, you must check the exact condition codes returned (the condition codes are defined in a system object library).

- CLI$_PRESENT—Element explicitly entered by the user
- CLI$_DEFAULTED—Element present by default
- CLI$_ABSENT—Element not entered by the user and not present by default
- CLI$_NEGATED—Element explicitly negated by the user

- CLI$_LOCPRES—Qualifier locally present (Section 7.3.3 describes positional qualifiers)
- CLI$_LOCNEG—Qualifier locally negated (Section 7.3.3 describes positional qualifiers)

In the following example, simply checking for success or failure would give undesired results. For example, if the user entered /FIX, success codes would be returned for both /ENTER (the default) and /FIX. (The example (an excerpt from a CLD file) provides for the possibility of changing the default qualifier in the CLD file without having to change the code.)

Excerpt from CLD File

```
QUALIFIER ENTER, DEFAULT
QUALIFIER FIX
QUALIFIER REPORT
DISALLOW ANY2 (ENTER, FIX, REPORT)
```

Excerpt from Program

```
INTEGER CLI$PRESENT
EXTERNAL CLI$_PRESENT,
2        CLI$_DEFAULTED
         .
         .
         .
IF ((CLI$PRESENT ('ENTER') .EQ. %LOC (CLI$_PRESENT))
2               .OR.
2  ((CLI$PRESENT ('ENTER') .EQ. %LOC (CLI$_DEFAULTED))
2               .AND.
2               (.NOT. CLI$PRESENT ('FIX'))
2               .AND.
2               (.NOT. CLI$PRESENT ('REPORT')))) THEN
         .
         .
         .
ELSE IF ((CLI$PRESENT ('FIX') .EQ. %LOC (CLI$_PRESENT))
2               .OR.
2  ((CLI$PRESENT ('FIX') .EQ. %LOC (CLI$_DEFAULTED))
2               .AND.
2               (.NOT. CLI$PRESENT ('ENTER'))
2               .AND.
2               (.NOT. CLI$PRESENT ('REPORT')))) THEN
         .
         .
         .
```

```
ELSE IF ((CLI$PRESENT ('REPORT') .EQ. %LOC (CLI$_PRESENT))
2                .OR.
2  ((CLI$PRESENT ('REPORT') .EQ. %LOC (CLI$_DEFAULTED))
2                 .AND.
2                 (.NOT. CLI$PRESENT ('ENTER'))
2                 .AND.
2                 (.NOT. CLI$PRESENT ('FIX')))) THEN
          .
          .
          .
END IF
```

If a keyword can be ambiguous, provide the full keyword path as described in Section 7.1.7.1. The following program (an excerpt from a CLD file) takes action only if the FORM1, FORM2, or FORM3 keyword is entered as a value to the /INPUT qualifier.

Excerpt from CLD File

```
QUALIFIER INPUT, VALUE (TYPE = FILE_TYPE)
QUALIFIER OUTPUT, VALUE (TYPE = FILE_TYPE)
DEFINE TYPE FILE_TYPE
KEYWORD FORM1
KEYWORD FORM2
KEYWORD FORM3
```

Excerpt from Program

```
INTEGER CLI$PRESENT
        .
        .
        .
IF (CLI$PRESENT ('INPUT.FORM1')) THEN
        .
        .
        .
ELSE IF (CLI$PRESENT ('INPUT.FORM2')) THEN
        .
        .
        .
ELSE IF (CLI$PRESENT ('INPUT.FORM3')) THEN
        .
        .
        .
END IF
```

7.3.2 Getting Element Values

Define the first argument to CLI$GET_VALUE as a character string whose value is the label of a parameter, qualifier, or keyword specified in the CLD file. Define the second argument as a character string large enough to receive the parameter, qualifier, or keyword value. Define the third (optional) argument as an INTEGER*2 integer to receive the length of the value returned in argument 2.

The following example (an excerpt from a CLD file) uses the value of parameter 1 in opening a file.

Excerpt from CLD File

```
PARAMETER P1
        LABEL = STATS_FILE
        VALUE (DEFAULT = "STATS.SAV")
```

Excerpt from Program

```
INTEGER STATUS,
2       CLI$GET_VALUE,
2       LIB$GET_LUN
INTEGER*2 FILE_NAME_SIZE
CHARACTER*255 FILE_NAME
INTEGER FILE_LUN
              .
              .
              .
STATUS = CLI$GET_VALUE ('STATS_FILE',
2                       FILE_NAME,
2                       FILE_NAME_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LIB$GET_LUN (FILE_LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
OPEN (UNIT=FILE_LUN,
2     FILE=FILE_NAME(1:FILE_NAME_SIZE),
2     STATUS='OLD',
2     FORM='UNFORMATTED')
```

If the value is not present (for example, if the value is not required and no default is supplied), a status of CLI$_ABSENT is returned.

If you are expecting a list of values, invoke CLI$GET_VALUE successively for each value in the list. Each time CLI$GET_VALUE is called, it returns one of the following as a status:

- SS$_NORMAL—If the value is the only or the last value in the list. The value is returned as argument 2.
- CLI$_COMMA—If the value is followed by a comma. The value is returned as argument 2.

- CLI$_CONCAT—If the value is followed by a plus sign. The value is returned as argument 2.
- CLI$_ABSENT—If the list is depleted. No value is returned.

If the separation of elements by commas or plus signs is not significant, you can simply check for CLI$_ABSENT on each invocation of CLI$GET_VALUE. The following example (an excerpt from a CLD file) reads a parameter list into an array (arbitrarily cutting the list off at 10 values).

Excerpt from CLD File

```
PARAMETER P1
        LABEL = STATS_FILE
        VALUE (LIST,
               DEFAULT = "STATS.SAV")
```

Excerpt from Program

```
INTEGER STATUS,
2       CLI$GET_VALUE,
2       FN_COUNT
INTEGER*2 FN_SIZE (10)
CHARACTER*255 FILE_NAME (10)
EXTERNAL CLI$_ABSENT
         .
         .
         .
! Get first value in list
STATUS = CLI$GET_VALUE ('STATS_FILE',
2                       FILE_NAME (1),
2                       FN_SIZE (1))
! Loop until CLI$GET_VALUE fails
DO WHILE (STATUS)
  ! Update count of values
  FN_COUNT = FN_COUNT + 1
  IF (FN_COUNT .LT. 10) THEN
    ! Get the next value in the list
    STATUS = CLI$GET_VALUE ('STATS_FILE',
2                           FILE_NAME (FN_COUNT + 1),
2                           FN_SIZE (FN_COUNT + 1))
  ELSE
    ! Force end of list at 10 values
    STATUS = %LOC (CLI$_ABSENT)
  END IF
END DO
! Make sure the bad status was the expected one
IF (STATUS .NE. %LOC (CLI$_ABSENT)) THEN
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

7.3.3 Determining the Position of a Qualifier

The positional environment of the parent command changes each time you invoke CLI$GET_VALUE for a parameter value: you are in the global environment before getting any parameter value, the P1 environment after getting P1, the P2 environment after getting P2, and so on. The rules for determining the presence of a qualifier in an environment are as follows:

- Global presence—If the user enters a positional (PLACEMENT = POSITIONAL) qualifier before entering any parameters, the qualifier is present (CLI$_PRESENT) in all environments except local environments explicitly specified by the user.
- Global negation—If the user negates a positional (PLACEMENT = POSITIONAL) qualifier before entering any parameters, the qualifier is negated (CLI$_NEGATED) in all environments except local environments explicitly specified by the user.
- Local presence—If the user enters a positional (PLACEMENT = POSITIONAL) or local (PLACEMENT = LOCAL) qualifier after a parameter, the qualifier is locally present (CLI$_LOCPRES) in the environment for that parameter.
- Local negation—If the user negates a positional (PLACEMENT = POSITIONAL) or local (PLACEMENT = LOCAL) qualifier after a parameter, the qualifier is locally negated (CLI$_LOCNEG) in the environment for that parameter.
- Default presence—If a positional (PLACEMENT = POSITIONAL) qualifier has the DEFAULT characteristic, the qualifier is present by default (CLI$_DEFAULTED) in all environments unless overridden by a user specification. If a local (PLACEMENT = LOCAL) qualifier has the DEFAULT characteristic, the qualifier is present by default (CLI$_DEFAULTED) in all parameter environments unless overridden by a user specification.

In most cases, your code should be secure if you simply check for the presence of the qualifier and obtain qualifier values in each of the parameter environments. The main care that you must exercise in working with positional qualifiers is to ensure that you are in the proper environment when looking at a qualifier. The following example (an excerpt from a CLD file) permits the user to enter a qualifier value (/FILETYPE = FORM1, /FILETYPE = FORM2, or /FILETYPE = FORM3) for each parameter by specifying the qualifier globally or by specifying it locally after each parameter. Note that by default each parameter environment gets a qualifier value of FORM1.

Excerpt from CLD File

```
PARAMETER P1, VALUE (REQUIRED)
              PROMPT = "Input file"
PARAMETER P2, VALUE (REQUIRED)
              PROMPT = "Output file"
QUALIFIER FILETYPE
          DEFAULT
          VALUE (TYPE = FILE_TYPE)
          PLACEMENT = POSITIONAL
DEFINE TYPE FILE_TYPE
KEYWORD FORM1, DEFAULT
KEYWORD FORM2
KEYWORD FORM3
```

Excerpt from Program

```
INTEGER STATUS,
2       CLI$PRESENT,
2       CLI$GET_VALUE
INTEGER*2 IF_SIZE,
2         OF_SIZE
CHARACTER*31 INPUT_FILE,
2            OUTPUT_FILE
CHARACTER*5 IF_TYPE,
2           OF_TYPE
STATUS = CLI$GET_VALUE ('P1',
2                       INPUT_FILE,
2                       IF_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = CLI$GET_VALUE ('FILETYPE',
2                       IF_TYPE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = CLI$GET_VALUE ('P2',
2                       OUTPUT_FILE,
2                       OF_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = CLI$GET_VALUE ('FILETYPE',
2                       OF_TYPE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.3.4 Examining the Command Line and Verb

You can obtain the contents of the command line exactly as entered by the user by passing the literal $LINE as the first argument to CLI$GET_VALUE. You can obtain the first four characters of the command name (no matter how many characters the user types) by passing the literal $VERB as the first argument to CLI$GET_VALUE. The following example returns the contents of the command line to the variable LINE and the first four characters of the command name to the variable VERB.

```
INTEGER STATUS,
2       CLI$GET_VALUE
INTEGER*2 LINE_SIZE
CHARACTER*1024 LINE
CHARACTER*4 VERB
STATUS = CLI$GET_VALUE ('$LINE',
2                       LINE,
2                       LINE_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = CLI$GET_VALUE ('$VERB',
2                       VERB)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4 Subcommands and Internal Parsing

The Command Definition Utility and associated CLI routines permit you to parse character strings obtained from sources other than the command line invoking your program.

7.4.1 Defining Subcommands

Define each subcommand in a DEFINE VERB block in a CLD file using the same rules as for any command. Compile the subcommands into an object command table by specifying the CLD file containing the subcommand definitions in a SET COMMAND command with the /OBJECT qualifier. Link the object command table to your program.

You can supply a module name for the command table by including a MODULE statement in the command definition file. If you do not include a MODULE statement, you can include a module name as a value to the /OBJECT qualifier, or let the module name default to the name of the CLD file (or the first CLD file processed).

The following example compiles three subcommands into an object module named INCOME_SUBCOMMANDS (the name of the object file is INCSUB.OBJ).

INCSUB.CLD

```
MODULE INCOME_SUBCOMMANDS
DEFINE VERB ENTER
DEFINE VERB FIX
  PARAMETER P1, LABEL = HOUSE_NO
                VALUE (REQUIRED, TYPE = $NUMBER)
                PROMPT = "House number"
  PARAMETER P2, LABEL = PERSONS_HOUSE
                VALUE (REQUIRED, TYPE = $NUMBER)
                PROMPT = "Persons per house"
  PARAMETER P3, LABEL = ADULTS_HOUSE
                VALUE (REQUIRED, TYPE = $NUMBER)
                PROMPT = "Adults per house"
  PARAMETER P4, LABEL = INCOME_HOUSE
                VALUE (REQUIRED, TYPE = $NUMBER)
                PROMPT = "Income per house"
DEFINE VERB REPORT
  QUALIFIER OUTPUT, VALUE (TYPE = $FILE,
                           DEFAULT = "INCOME.RPT")
```

```
$ SET COMMAND/OBJECT INCSUB
```

The object file containing the command definition module must be linked to your main program and other program units.

7.4.2 Parsing Subcommands

A program must explicitly parse a subcommand by invoking the Run-Time Library routine CLI$DCL_PARSE. Specify the arguments to CLI$DCL_PARSE as follows:

- Input line or 0 (argument 1)—If you are using argument 4 to read the entire command, specify argument 1 as a 0. Pass the argument by value. If you have already read (or otherwise obtained) the command or part of the command, specify the character string containing the command as argument 1 (and pass it by descriptor, the default mechanism).
- Subcommand table (argument 2)—Specify the name of the module containing the subcommand table. The module name must be defined as external.
- Required parameter routine (argument 3)—If the subcommand definitions contain any required parameters, specify the name of a routine to prompt for and read the parameters in case the user does not enter them on the command line. The routine can be LIB$GET_INPUT or any other routine that takes the same arguments and returns a status code. The routine name must be defined as external.

- Command line routine (argument 4)—Specify the name of a routine to read the entire command (if you did not obtain the command and pass it to CLI$DCL_PARSE as argument 1) or continuation command lines. The routine can be LIB$GET_INPUT or any other routine that takes the same three arguments and returns a status code. The routine name must be defined as external.
- Command line prompt (argument 5)—If you specify argument 4, specify a character string to be written as a prompt to obtain the command or the rest of the command.

The following example reads (using LIB$GET_INPUT and the prompt INCOME>) and parses a subcommand, and performs various actions depending on the subcommand that the user enters. The program processes subcommands until the user presses CTRL/Z.

```
INTEGER STATUS,
2       CLI$DCL_PARSE,
2       CLI$GET_VALUE
INCLUDE '($RMSDEF)'
INCLUDE '($STSDEF)'
EXTERNAL INCOME_SUBCOMMANDS,
2        LIB$GET_INPUT
CHARACTER*4 SUBCOMMAND_NAME
STATUS = LIB$PUT_OUTPUT
2 ('Subcommands: ENTER --- FIX --- REPORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LIB$PUT_OUTPUT ('Press CTRL/Z to exit')
! Get first subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                       INCOME_SUBCOMMANDS, ! CLD module
2                       LIB$GET_INPUT,      ! Parameter routine
2                       LIB$GET_INPUT,      ! Command routine
2                       'INCOME> ')         ! Command prompt
```

```
! Do it until user presses CTRL/Z
DO WHILE (STATUS .NE. RMS$_EOF)
  ! If no error on CLI$DCL_PARSE
  IF (STATUS) THEN
    ! Get name of subcommand --- always 4 characters
    STATUS = CLI$GET_VALUE ('$VERB',
2                           SUBCOMMAND_NAME)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Perform appropriate action
    IF (SUBCOMMAND_NAME .EQ. 'ENTE') THEN
            .
            .
            .
    ELSE IF (SUBCOMMAND_NAME .EQ. 'FIX ') THEN
            .
            .
            .
    ELSE IF (SUBCOMMAND_NAME .EQ. 'REPO') THEN
            .
            .
            .
    END IF
    ! Do not signal warning again
  ELSE IF (IBITS (STATUS, 0, 3) .NE. STS$K_WARNING) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Get another subcommand
  STATUS = CLI$DCL_PARSE (%VAL (0),
2                         INCOME_SUBCOMMANDS,
2                         LIB$GET_INPUT,
2                         LIB$GET_INPUT,
2                         'INCOME> ')
END DO
END
```

If the user makes a syntax error in the subcommand, CLI$DCL_PARSE both signals the error and returns it as a status code. The severity of a syntax error is a warning, so that the default action (if you do not establish your own condition handler) is to type the message and continue execution. You can trap a warning by comparing the lower 3 bits of the status code to STS$K_WARNING (defined in $STSDEF). The example shown above refrains from explicitly calling LIB$SIGNAL if an error returned by CLI$DCL_PARSE is a warning, so that the error message is not typed twice.

If an error occurs while the routine specified by argument 4 is reading the command line, the error status is passed along as the status returned by CLI$DCL_PARSE. In the example shown above, the program can expect to receive the status RMS$_EOF (defined in $RMSDEF) from CLI$DCL_PARSE (CLI$DCL_PARSE receives the status from LIB$GET_INPUT) when the user presses CTRL/Z.

If you write your own routine for argument 3 or 4, write it as a FUNCTION subprogram and return a status code as the function value (return SS$_NORMAL to continue execution of the program). Specify the dummy arguments for the routine as follows:

- Argument 1—Assumed-length character string in which you must place the character string that you read
- Argument 2—Assumed-length character string containing a prompt; this value is passed to you from argument 5 of CLI$DCL_PARSE
- Argument 3—INTEGER*2 integer in which you must place the size of the character string that you read

The following example demonstrates the necessary declarations.

```
INTEGER FUNCTION GETINPUT (INPUT,   ! Returned
2                          PROMPT,  ! Passed
2                          SIZE)    ! Returned
CHARACTER*(*) INPUT,
2             PROMPT
INTEGER*2     SIZE
INTEGER STATUS
      .
      .
      .
GETINPUT = STATUS
END
```

7.4.3 Dispatching to a Subprogram

You can invoke a subprogram depending on the subcommand (that is, the verb) entered by the user. You must specify the name of the subprogram to be invoked for a particular subcommand with a ROUTINE statement in the DEFINE VERB block for the subcommand. In your program, simply invoke CLI$DISPATCH after you invoke CLI$DCL_PARSE. The following example (a CLD file) causes one of the subprograms ENTER, FIX, and REPORT to be invoked depending on which subcommand is entered by the user.

```
MODULE INCOME_SUBCOMMANDS
DEFINE VERB ENTER
  ROUTINE ENTER
DEFINE VERB FIX
  ROUTINE FIX
       .
       .
       .
DEFINE VERB REPORT
  ROUTINE REPORT
  QUALIFIER OUTPUT, VALUE (TYPE = $FILE,
                           DEFAULT = "INCOME.RPT")
```

```
PROGRAM INCOME
INTEGER STATUS,
2       CLI$DCL_PARSE,
2       CLI$DISPATCH
INCLUDE '($RMSDEF)'
INCLUDE '($STSDEF)'
EXTERNAL INCOME_SUBCOMMANDS,
2        LIB$GET_INPUT
STATUS = LIB$PUT_OUTPUT
2 ('Subcommands: ENTER --- FIX --- REPORT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = LIB$PUT_OUTPUT
2 ('Press CTRL/Z to exit')
! Get first subcommand
STATUS = CLI$DCL_PARSE (%VAL (0),
2                       INCOME_SUBCOMMANDS, ! CLD module
2                       LIB$GET_INPUT,      ! Parameter routine
2                       LIB$GET_INPUT,      ! Command routine
2                       'INCOME> ')         ! Command prompt
! Do it until user presses CTRL/Z
DO WHILE (STATUS .NE. RMS$_EOF)
  ! If no error on CLI$DCL_PARSE
  IF (STATUS) THEN
    ! Dispatch depending on subcommand
    STATUS = CLI$DISPATCH ()
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    ! Do not signal warning again
  ELSE IF (IBITS (STATUS, 0, 3) .NE. STS$K_WARNING) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Get another subcommand
  STATUS = CLI$DCL_PARSE (%VAL (0),
2                         INCOME_SUBCOMMANDS,
2                         LIB$GET_INPUT,
2                         LIB$GET_INPUT,
2                         'INCOME> ')
END DO
END
INTEGER FUNCTION ENTER ()
INTEGER STATUS
        .
        .
        .
ENTER = STATUS
END
INTEGER FUNCTION FIX ()
INTEGER STATUS
        .
        .
        .
FIX = STATUS
END
```

```
INTEGER FUNCTION REPORT ()
INTEGER STATUS
        .
        .
        .
REPORT = STATUS
END
```

If you define your subprogram as an integer function, the return value is used by CLI$DISPATCH as a status code. Otherwise, CLI$DISPATCH does not return a proper status code.

You can pass one optional longword from CLI$DISPATCH to your subprogram. All other data must be passed by defining common areas.

7.4.4 Returning to the Invoking Command

When you invoke CLI$DCL_PARSE for a subcommand, further CLI$PRESENT and CLI$GET_VALUE invocations apply to the subcommand. You no longer have access to the command that invoked your program from DCL command level. You can reparse the command that invoked your program by invoking CLI$DCL_PARSE specifying zeros for the first two arguments and omitting the remaining arguments. You must pass the first argument by value.

8 User Input/Output

Typically, you set up your program so that the user is the invoker. The user starts the program by naming the DCL command with which it is associated (see Chapter 7) or by naming the image file (for example, in the RUN command). The user's input and output devices are defined by the logical names SYS$INPUT and SYS$OUTPUT, which are initially equivalenced as follows:

Logical Name	User Mode	Equivalence Device or File
SYS$INPUT	Interactive	Terminal on which user is logged in
	Batch job	Data lines following the invocation of the program
	Command procedure	Data lines following the invocation of the program
SYS$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.

Alternatively, you can design your program to get input from and put 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 do 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. Chapter 9 describes file input/output.

8.1 Conversational I/O

Usually, you can request information from, or write information to, the user with little regard for formatting. For such conversational I/O, use the Run-Time Library routines LIB$GET_INPUT and LIB$PUT_OUTPUT. If programming in FORTRAN, the FORTRAN I/O statements READ, ACCEPT, WRITE, PRINT, and TYPE can also be used. For the I/O statements available in your language, see your language-specific programming manual.

To provide fancy (complex) screen displays for input or output, use the screen management facility described in Section 8.2.

8.1.1 Device Selection

The Run-Time Library procedures 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 procedures; you need only invoke the procedure to access the I/O unit (device or file) associated with the logical name. You cannot use the procedures to access an I/O unit other than the one associated with SYS$INPUT or SYS$OUTPUT.

With FORTRAN I/O, you can use the OPEN statement to specify an I/O unit, or omit the OPEN statement and use an implied I/O unit.

To identify an I/O unit for FORTRAN, you use a logical unit number—an integer value of your choice. Specify the logical unit number when you open the I/O unit (UNIT keyword of the OPEN statement). To read or write to that I/O unit, specify the logical unit number in the appropriate FORTRAN I/O statement (UNIT keyword of the I/O statement). When you use an implied I/O unit, you do not specify a logical unit number; FORTRAN uses an implied logical unit number.

If more than one person is working on a program, you should generate logical unit numbers with the Run-Time Library procedure LIB$GET_LUN, rather than choosing your own values, to ensure that the number is unique among all logical unit numbers used in the program.

8.1.1.1 Implied Unit for FORTRAN I/O

In FORTRAN I/O statements, if you specify UNIT as an asterisk (*) or omit UNIT, input statements use an implied unit of SYS$INPUT and output statements use an implied unit of SYS$OUTPUT. (Not all FORTRAN I/O statements allow you to omit UNIT; for a complete description of each FORTRAN statement, see *Programming in VAX FORTRAN*.) In addition, the I/O units are associated with special FORTRAN logical names. The following table details the associations of FORTRAN implied units with FORTRAN and system logical names.

FORTRAN Statement	FORTRAN	System
READ (UNIT=*,...) list	FOR$READ	SYS$INPUT
READ fmt, list	FOR$READ	SYS$INPUT
ACCEPT fmt, list	FOR$ACCEPT	SYS$INPUT
WRITE (UNIT=*,...) list	FOR$PRINT	SYS$OUTPUT
PRINT fmt, list	FOR$PRINT	SYS$OUTPUT
TYPE fmt, list	FOR$TYPE	SYS$OUTPUT

A person using your program can redirect input and output either by redefining SYS$INPUT and SYS$OUTPUT, or by defining one of the FORTRAN logical names. The FORTRAN logical name (when it is defined) takes precedence over the system logical name.

When you use an implied I/O unit, do not open or close the unit. Simply issue the appropriate I/O statement. Since FORTRAN opens an implied I/O unit using the default characteristics of the OPEN statement (including ACCESS = 'SEQUENTIAL' and CARRIAGECONTROL = 'FORTRAN'), you must remember to specify the first output character as the FORTRAN carriage control character and not a data character. The following example demonstrates the use of a WRITE statement to write to SYS$OUTPUT.

```
INTEGER       OUTPUT_SIZE
CHARACTER*512 OUTPUT
 .
 .
 .
! The leading space is for FORTRAN carriage control,
! the default characteristic for formatted, sequential files
WRITE (UNIT=*,
2      FMT='(2A)') ' ', OUTPUT (1:OUTPUT_SIZE)
```

8.1.1.2 Explicit Unit for FORTRAN I/O

To write to a device other than the one associated with SYS$INPUT or SYS$OUTPUT, you must name the device (physical or logical name) in an OPEN statement and use the logical unit number of the OPEN statement in subsequent READ and WRITE statements. The following example writes to the device named $TERMINAL1.

```
INTEGER       STATUS,
2             OUTPUT_SIZE
CHARACTER*512 OUTPUT
  .
  .
  .
! Open output file with list carriage control
OPEN (UNIT=1,
2     FILE='$TERMINAL1',
2     STATUS='UNKNOWN',
2     CARRIAGECONTROL='LIST')
  .
  .
  .
WRITE (UNIT=1,
2      FMT='(A)') OUTPUT (1:OUTPUT_SIZE)
```

You can also access SYS$INPUT or SYS$OUTPUT by naming the unit in an OPEN statement and using the logical unit number of the OPEN statement in subsequent READ and WRITE statements. This technique is useful if you want to assign other than the default characteristics to the I/O unit.

In the OPEN statement, specify the FILE attribute as FILE = 'SYS$INPUT' (for read operations) or FILE = 'SYS$OUTPUT' (for write operations). Specify the STATUS attribute as 'OLD'. The following example explicitly opens SYS$OUTPUT to use list carriage control (rather than FORTRAN carriage control, the default).

```
INTEGER       STATUS,
2             OUTPUT_SIZE
CHARACTER*512 OUTPUT
  .
  .
  .
! Open SYS$OUTPUT with list carriage control
OPEN (UNIT=1,
2     FILE='SYS$OUTPUT',
2     STATUS='OLD',
2     CARRIAGECONTROL='LIST')
  .
  .
  .
! Write a line to SYS$OUTPUT
WRITE (UNIT=1,
2      FMT='(A)') OUTPUT (1:OUTPUT_SIZE)
```

If you specify a FORTRAN logical name (FOR$READ, FOR$ACCEPT, FOR$PRINT, or FOR$TYPE) in the OPEN statement, it must be defined before the program is invoked or an error (error in file name) occurs.

8.1.2 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 RETURN.

If you are reading character data, LIB$GET_INPUT is a simple way of prompting for and reading the data. If you are reading noncharacter data, FORTRAN I/O is preferable since it allows you to translate the data to a format of your choice. (Format specifications for the FORTRAN I/O statements are described in *Programming in VAX FORTRAN*.) For input, FORTRAN I/O offers the ACCEPT statement, which reads data from SYS$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 FORTRAN I/O does not necessarily cause an error, but does truncate your data.

8.1.2.1 LIB$GET_INPUT

LIB$GET_INPUT places the characters read in a variable of your choice. You must define the variable as character in type. Optionally, LIB$GET_INPUT places the number of characters read in another variable of your choice which must be defined as INTEGER*2 in type. On terminal input, LIB$GET_INPUT optionally writes a prompt before reading the input. The prompt is suppressed automatically for a nonterminal operation.

The following example reads a line of input using LIB$GET_INPUT.

```
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. For 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.

8.1.2.2 ACCEPT Statement

The FORTRAN ACCEPT statement reads data from FOR$ACCEPT (if it is defined) or SYS$INPUT. The data, which must be read using sequential access mode, is transferred from the I/O unit to a variable or variables of your choice. You can specify the format of the data or use list-directed I/O by specifying the format as an asterisk (*). The following example uses list-directed I/O to read a numeric variable from SYS$INPUT.

```
INTEGER*4    NUMBER
          .
          .
          .
ACCEPT *, NUMBER
```

8.1.2.3 READ Statement

The FORTRAN READ statement transfers data from an I/O unit to a variable or variables of your choice. By default, an input record is restricted to 132 characters in length. You can override the default by explicitly opening the input unit and specifying the RECL specifier. Overflowing the maximum length causes the fatal error FOR$_ERRDURREA (defined in $FORDEF).

For character input (that is, if you do not convert the data on input), the following guidelines are suggested:

- Input size—Use the Q format specification to obtain the number of characters in the input record. Define the variable associated with the Q specification as an integer data type.
- Input data—Use the A format specification to obtain the actual data. Define the variable associated with the A specification as a character data type. Specify the variable as a character string whose bounds are 1 and the size read by the Q specification.
- Prompt—To prompt, first issue a WRITE statement using the A format specification for the actual prompt, followed by the dollar sign ($) format specification to suppress the carriage return (if you want the input data to

appear on the same line as the prompt). Alternatively, you can omit the dollar sign format specification and use the dollar sign as a FORTRAN carriage control character by making it the first character of the prompt string.

Note

You can format the READ statement to process more than one record; for details, see *Programming in VAX FORTRAN*.

The following example issues a prompt to SYS$OUTPUT and obtains an input value from SYS$INPUT.

```
INTEGER       INPUT_SIZE
CHARACTER*512 INPUT
WRITE (UNIT=*,
2      FMT='(A,$)') ' Input value: '
READ (UNIT=*,
2     FMT='(Q,A)') INPUT_SIZE,
2                  INPUT (1:INPUT_SIZE)
! The leading space in the prompt is the FORTRAN carriage
! control character, which must be included unless you
! open SYS$INPUT and specify carriage control other than
! FORTRAN. The trailing space is so a space appears on
! the terminal after the colon.
```

In further references to input character data, you should specify the appropriate substring or array elements rather than the entire character variable. For example, if you read characters into a character 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.

If you are reading numeric or logical data, you can perform the conversion as part of the I/O operation. You should check for the error condition FOR$_INPCONERR (defined in $FORDEF) in case the character input was not suitable for conversion to a number. The following example reads the input value into an integer variable.

```
INTEGER   WHOLE_NUMBER,
2         IOSTAT,
2         STATUS,
2         IO_OK
PARAMETER (IO_OK = 0)
INCLUDE   '($FORDEF)'
WRITE (UNIT=*,
2      FMT='(A,$)') ' Input value: '
READ (UNIT=*,
2     IOSTAT=IOSTAT,
2     FMT='(BN,I)') WHOLE_NUMBER
! Error loop in case the user types in a noninteger value
DO WHILE (IOSTAT .NE. IO_OK)
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .EQ. FOR$_INPCONERR) THEN
    WRITE (UNIT=*,
2          FMT='(A,$)') ' Oops. Try again: '
    READ (UNIT=*,
2         IOSTAT=IOSTAT,
2         FMT='(BN,I)') WHOLE_NUMBER
  ELSE
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END DO
```

8.1.3 Getting Many 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 $RMSDEF) when end-of-file occurs. FORTRAN generates the error FOR$_ENDDURREA (defined in $FORDEF) when end-of-file occurs. On a FORTRAN read operation, you can also detect end-of-file by checking the I/O return status (IOSTAT) for a negative value or by using an END specifier in the READ statement.

The following example uses a FORTRAN READ statement in a loop to read a sequence of integers from SYS$INPUT.

```
! 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
! 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
```

8.1.4 Writing Output

If you are writing character data, LIB$PUT_OUTPUT is a simple way of writing the data. If you are writing noncharacter data, FORTRAN I/O is preferable since it allows you to translate the data to a format of your choice. (Format specifications for the FORTRAN I/O statements are described in *Programming in VAX FORTRAN*.) For output, FORTRAN I/O offers the TYPE or PRINT statement, which writes data to SYS$INPUT, and the READ statement, which writes data to an I/O unit of your choice.

8.1.4.1 LIB$PUT_OUTPUT

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 character in type. 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))
```

8.1.4.2 WRITE Statement

The FORTRAN WRITE statement writes one or more output records to an I/O unit depending upon the format. For character data, use the A format specification. If you specify the output data as substrings, you do not need to attach a number to the A format specification. If you use FORTRAN carriage control, be sure that the first character of the output record is the carriage control character (see Section 8.1.4.7) and not data. The following example writes a character expression.

```
CHARACTER*31 ANSWER
INTEGER      ANSWER_SIZE
      .
      .
      .
! Write answer
! Leading space is for FORTRAN carriage control
WRITE (UNIT=*,
2      FMT='(A)') ' Answer: ' // ANSWER (1:ANSWER_SIZE)
```

As an alternative to concatenating the output data, you can specify the output data as a comma-separated list of constants and variables, and use list-directed formatting or provide a format specification for each element of the list. You can also include constants within the format specification rather than within the list of output data. The following WRITE statements are alternatives to the example shown above.

Example 1

```
WRITE (UNIT=*,
2      FMT=*) 'Answer: ', ANSWER (1:ANSWER_SIZE)
```

Example 2

```
WRITE (UNIT=*,
2      FMT='(2A)') ' Answer: ', ANSWER (1:ANSWER_SIZE)
```

Example 3

```
WRITE (UNIT=*,
2      FMT='('' Answer: '',A)') ANSWER (1:ANSWER_SIZE)
```

8.1.4.3 Formatting Numeric Output

You can convert numeric data as part of the output operation, as demonstrated here:

```
INTEGER*4 ANSWER
        .
        .
        .
WRITE (UNIT=*,
2      FMT='(A,I)') ' Answer: ',
2                    ANSWER
```

However, the appearance of the data may not be what you desire depending on how the format specification converts the number. For example, the I specification places the character representation of the number in a field of 12 positions, right justified, and padded on the left with blanks. If ANSWER equals 31, the above code would write the following line:

```
Answer:          31
```

If you know exactly how many character positions the number requires, you can include that value as part of the specification. For example, you could use a specification of I2 instead of I. A more generic solution would be to convert the number to a character string internally using the default specification, then strip the leading blanks before writing the value.

```
INTEGER*4    ANSWER,
2            START
CHARACTER*12 STRING
         .
         .
         .
! Convert number to character string
WRITE (UNIT=STRING,
2      FMT='(I)') ANSWER
! Locate first nonblank character
START = 1
DO WHILE (STRING (START:START) .EQ. ' ')
  START = START + 1
END DO
! Write substring starting at first nonblank character
WRITE (UNIT=*,
2      FMT='(2A)') ' Answer: ', STRING (START:12)
```

8.1.4.4 Multiple Output Records

You can place the output data in more than one record by separating format specifications with a slash (/). The following example writes the constant on one line and the variable on the next line.

```
WRITE (UNIT=*,
2      FMT='(A/1X,A)') ' Answer: ', ANSWER (1:ANSWER_SIZE)
```

8.1.4.5 TYPE Statement

The FORTRAN TYPE statement transfers one or more records to SYS$OUTPUT (FOR$TYPE, if it is defined). You can specify the format of the data or use list-directed I/O by specifying the format as an asterisk (*). A list-directed I/O record is automatically constructed with a leading space; therefore, do not specify a FORTRAN carriage control character as the first character of output. The following example writes a character constant and a numeric variable to SYS$OUTPUT.

```
INTEGER*4 ANSWER
        .
        .
        .
TYPE *, 'Answer: ', ANSWER
```

8.1.4.6 Processing Arrays and Records

The FORTRAN WRITE statement permits you to specify unqualified array names in formatted or unformatted I/O. However, unqualified record names can be used only in unformatted I/O. When using formatted I/O with records, you must treat each record field as a separate variable. The following program segment uses formatted I/O to write the whole of array STOCK_NUMBER and information from the ITEM record.

```
INTEGER*4 STOCK_NUMBER(10)
STRUCTURE /ITEM_DESCR/
 INTEGER*4    CODE
 CHARACTER*80 DESCRIPTION
 INTEGER*4    D_LEN
END STRUCTURE
RECORD /ITEM_DESCR/ ITEM
                .
                .
                .
WRITE (UNIT=*,
2      FMT='(10I4)') STOCK_NUMBER
WRITE (UNIT=*,
2      FMT='(I10,''   '',A)') ITEM.CODE,
2                              ITEM.DESCRIPTION (1:ITEM.D_LEN)
```

The FORTRAN WRITE statement permits you to process a series of data elements with an implied DO loop. An implied DO loop is particularly useful for processing an array. The following example writes one line for each element in the ITEMS array starting with element 1 and ending with the element whose position is the value of ITEM_COUNT.

```
STRUCTURE /ITEM_DESCR/
 INTEGER*4    CODE
 CHARACTER*80 DESCRIPTION
 INTEGER*4    D_LEN
END STRUCTURE
RECORD /ITEM_DESCR/ ITEMS (255)
INTEGER*4 ITEM_COUNT
WRITE (UNIT=*,
2      FMT='(I10,''   '',A)')
2                 (ITEMS(I).CODE,                        ! Part 1
2                  ITEMS(I).DESCRIPTION (1:ITEMS(I).D_LEN),
2                  I = 1, ITEM_COUNT)                    ! Part 2
```

Note that the I/O list must be enclosed in parentheses. The first part of the I/O list consists of the data elements being written; the second part of the I/O list is an iterative DO loop specification. In the example, the variable I serves as the control variable. The control variable can be used in the first part of the I/O list (as it is in the example) but cannot be modified.

The above implied DO loop is equivalent to the following explicit DO loop:

```
DO I = 1, ITEM_COUNT
  WRITE (UNIT=*,
2        FMT='(I10,''   '',A)')
2                 (ITEMS(I).CODE,
2                  ITEMS(I).DESCRIPTION (1:ITEMS(I).D_LEN))
END DO
```

8.1.4.7 Carriage Control

The standard carriage control operations are as follows:

- Single spacing—Writing of the output record is preceded by a line feed and carriage return, and is terminated by a carriage return. Single spacing occurs if you use list carriage control. If you use FORTRAN carriage control (the default for formatted I/O), specify the control character (the first character of the record) as a blank.
- Double spacing—Writing of the output is preceded by two line feeds and a carriage return, and is terminated by a carriage return. Double spacing occurs if you use FORTRAN carriage control and specify the control character as 0 (zero).
- Form feed—Writing of the output is preceded by a form feed and a carriage return, and is terminated by a carriage return. Use FORTRAN carriage control and specify the control character as the character 1.

- Overprinting—Writing of the output is preceded by a carriage return (no form feed or line feed), and is terminated by a carriage return. Use FORTRAN carriage control and specify the control character as a plus sign (+).
- Prompt—Writing of the output is preceded by a line feed and carriage return. No termination carriage return is output. Use FORTRAN carriage control and specify the control character as a dollar sign ($) or use the dollar sign format specification.

List carriage control occurs by default when you use the WRITE or TYPE statement without a format specification. FORTRAN carriage control occurs by default when you use the WRITE or TYPE statement with a format specification. You can override the default in the WRITE statement by explicitly opening the output device and specifying the CARRIAGECONTROL specifier.

8.2 Screen Management

The suggested tools for managing the appearance of the terminal screen are the SMG Run-Time Library routines, which provide a simple, device-independent interface to the terminal. The 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:

1. Create a pasteboard—A pasteboard is a logical two-dimensional area on which you place virtual displays. Use the SMG$CREATE_PASTEBOARD routine to create a pasteboard and associate it with a physical device. You refer to the pasteboard and SMG performs the necessary I/O to the device.
2. Create a virtual display—A virtual display is a logical two-dimensional area in which you place the information to be displayed. Use the SMG$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 or not that display is currently pasted to a pasteboard.

The following example 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.

```
! 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 8.3 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 *VAX/VMS Run-Time Library Routines Reference Manual*.

8.2.1 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.

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))
```

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.

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.

```
INTEGER*4 WIDTH,
2         COLOR
INCLUDE   '($SMGDEF)'
```

```
! 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))
```

8.2.2 Virtual Displays

Using the SMG routines, you write to virtual displays which are logically configured as rectangles. 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 one.

8.2.2.1 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 will occupy 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
```

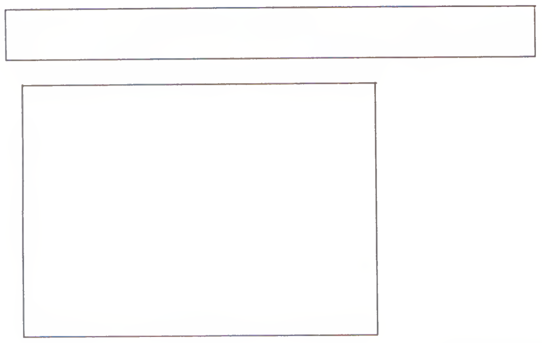
8.2.2.2 Pasting Virtual Displays

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. The following example defines two virtual displays and pastes them to one pasteboard.

```
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))
```

```
! 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))
                    .
                    .
                    .
```

The following figure shows the resultant screen.



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 8.2.2.3, 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 two-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.

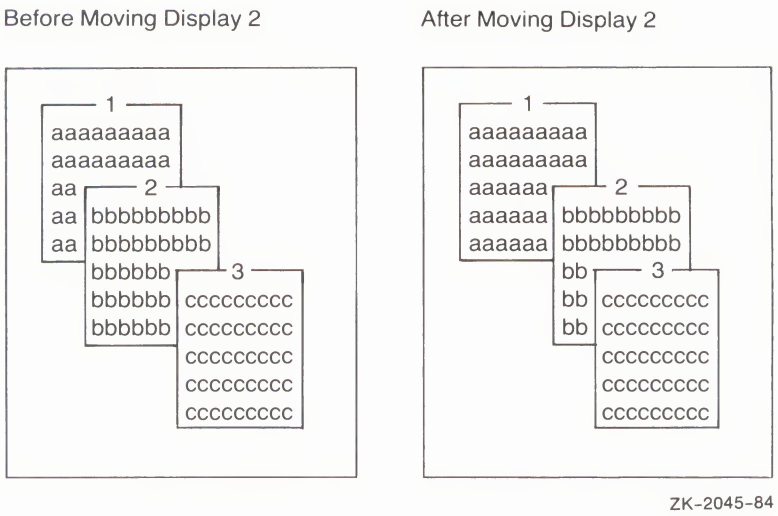
```
  STATUS = SMG$CHECK_FOR_OCCLUSION (STATS_VDID,
2                                   PBID,
2                                   OCCLUDE_STATE)
! OCCLUDE_STATE must be defined as INTEGER*4
  IF (OCCLUDE_STATE) THEN
    STATUS = SMG$UNPASTE_VIRTUAL_DISPLAY (SUM_VDID,
2                                         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
```

8.2.2.3 Rearranging Virtual Displays

Pasted displays can be rearranged by moving or repasting.

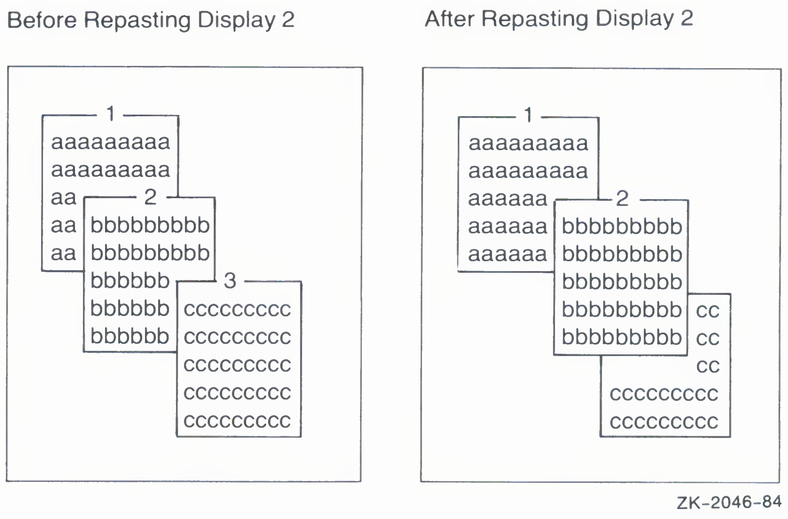
- Moving—To move a display, use the SMG$MOVE_VIRTUAL_DISPLAY routine. The following example moves display 2; the figure following the example shows the screen before and after the statement executes.

```
STATUS = SMG$MOVE_VIRTUAL_DISPLAY (VDID,
2                                  PBID,
2                                  5,
2                                  10)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```



- Repasting—To repaste a display, use the SMG$REPASTE_VIRTUAL_DISPLAY routine. The following example repastes display 2; the figure following the example 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))
```



8.2.2.4 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.

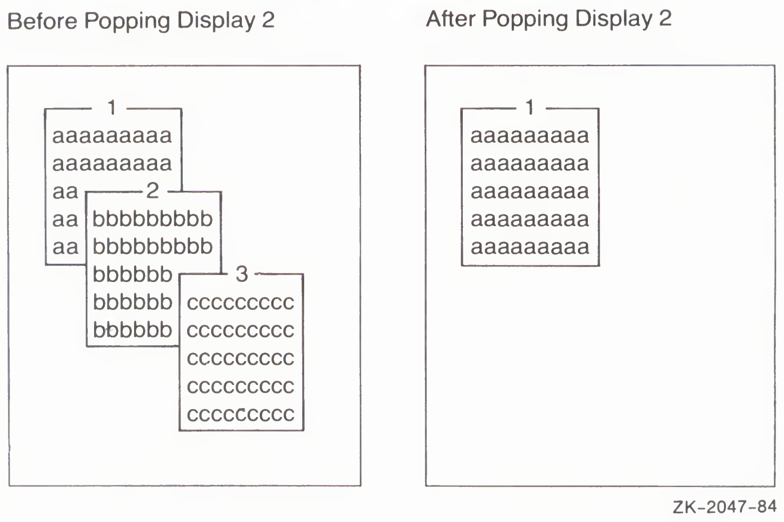
```
STATUS = SMG$UNPASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                     PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- 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; the figure following the example 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))
```



8.2.2.5 Modifying a Virtual Display

The screen management facility provides two routines for modifying the characteristics of an existing virtual display: SMG$CHANGE_VIRTUAL_DISPLAY, which allows you to change the size, video attributes, or border of a display; and SMG$CHANGE_RENDITION, which allows you to change the video attributes of a portion of a display.

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$CHANGE_RENDITION uses three sets of video attributes to determine the attributes to apply to the specified portion of the display: the display's default video attributes, the attributes specified by the **rendition-set** argument of SMG$CHANGE_RENDITION, and the attributes specified by the **rendition-complement** argument of SMG$CHANGE_RENDITION. The following table shows the result of each possible combination.

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 **rendition-complement** but not in **rendition-set** specifying that SMG 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.

8.2.3 Writing

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 display are visible only if the display is pasted to a pasteboard.

8.2.3.1 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.

8.2.3.2 Writing Data Character by Character

If you are writing character by character (see Section 8.2.3.3 for line-oriented output), SMG$PUT_CHARS is a simple, precise tool. The routine places exactly the specified characters on the screen, starting at a specified position in a virtual display (which defaults to the current cursor position). Anything currently in the positions written to is overwritten; no other positions on the screen are affected. Convert numeric data to character data with a FORTRAN internal WRITE operation before invoking SMG$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.

To insert characters rather than overwriting 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.

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 four character routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the characters being written. The explanation of the SMG$CHANGE_RENDITION routine in Section 8.2.2.5 discusses how to use **rendition-set** and **rendition-complement**.

8.2.3.3 Writing Data Line by Line

The routines SMG$PUT_LINE and SMG$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. Both routines permit you to scroll forward (up); SMG$PUT_WITH_SCROLL permits you to scroll backward (down). SMG$PUT_LINE permits other than single spacing.

The following example writes lines from a buffer to a display area. The output is scrolled forward if the buffer contains more lines than the display area.

```
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
```

The next example scrolls the output backward.

```
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
```

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.

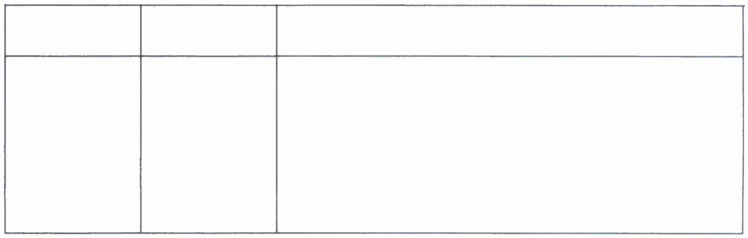
To insert text rather than overwriting the current contents of the screen, use the routine SMG$INSERT_LINE. Existing lines are shifted up or down (you specify) to open space for the new text. If the text is longer than a single line, the excess characters are truncated or wrapped (you specify).

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-size characters on a line.

All four line routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the text being written. The explanation of the SMG$CHANGE_RENDITION routine in Section 8.2.2.5 discusses how to use **rendition-set** and **rendition-complement**.

8.2.3.4 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 the following figure in the statistics display area (an area of 10 rows by 55 columns).



ZK-2048-84

You might write the following code:

```
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))
```

8.2.3.5 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.

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))
```

8.2.4 Reading

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. Section 8.3 describes how to use the input routines without the output routines. This section assumes that you are using the input routines with the output routines.

When using the SMG input routines with the SMG output routines, always specify the optional 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, may not be occluded, and may not have any other display to its right; input begins at the current cursor position but the cursor must be in column 1.

8.2.4.1 Reading from a Display

You can read the contents of the screen using the routine SMG$READ_FROM_DISPLAY. By default, the read operation reads all of the characters from the current cursor position to the end of that line. The third argument of SMG$READ_FROM_DISPLAY allows you to choose the starting point of the read operation by providing a set of "terminators." 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 the beginning of the line). You must calculate the length of the character string read 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
```

The SMG$READ_FROM_DISPLAY routine provides a simple input mechanism for menu driven applications. The sample program in Section 8.2.4.3 creates a simple menu, allows a user to use the numeric keys on the keypad to position the cursor at the beginning of a menu entry, and reads the entry when the user presses RETURN.

8.2.4.2 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 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 and 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.

The following example reads input until CTRL/Z is pressed.

```
! 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)
! 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 may not scroll as expected. When scrolling a mix of input and output, you can prevent possible problems by using SMG$PUT_CHARS.

8.2.4.3 Reading from the Keypad

To read from the keypad in keypad mode (that is, the user presses 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.

The following example 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$SET_CURSOR_REL is called instead of invoked as a function because you do not want to abort the program on an error. (The error would be attempting to move the cursor out of the display area and you just want the cursor not to move if this error occurs.) The read operation is also modified with TRM$M_TM_PURGE to prevent the user from getting ahead of the cursor.

```
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$SIGNAL (%VAL (STATUS))
! Read character from keyboard
MODIFIERS = TRM$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$READ_STRING (KID,
2                            INPUT_STRING,
2                            ,
2                            6,
2                            MODIFIERS,
2                            ,
2                            ,
2                            ,
2                            TERMINATOR)
END DO
! Read menu entry and process
!     Guidelines for reading from the display
!     are in Section 8.2.4.1.
STATUS = SMG$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
```

8.2.4.4 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

- 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 RETURN 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.

The following example defines the keys 1 through 9 on the keypad and permits the user to temporarily change state by pressing the PF1 key (the gold key). Pressing 1 on the keypad is equivalent to typing 1000 and pressing RETURN. Pressing PF1 and then 1 on the keypad is equivalent to typing 10000 and pressing return.

```
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$SIGNAL (%VAL (STATUS))
! Pressing KP1 through KP9 in the BYTEN state is like
! typing 10000 through 90000 and pressing return
STATUS = SMG$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.

In order to use keypad keys 0 through 9, the keypad must be in application mode. (Use the /APPLICATION qualifier of the DCL command SET TERMINAL; see the *VAX/VMS DCL Dictionary* for details.)

8.2.5 Controlling Screen Updates

If your program needs to make a number of changes to a virtual display, you may 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.

8.2.6 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. Observe the following guidelines:

- Mixing SMG 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 non-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 below:

```
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 also 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)
```

8.3 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 have a mechanism for reading that character. You can also control such things as echoing, time allowed for input, and whether or not 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 *VAX/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 8.2 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:

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.

8.3.1 CTRL/C and CTRL/Y Interrupts

The QIO system services enable you to detect a CTRL/C or CTRL/Y interrupt (the terminal user types CTRL/C or CTRL/Y) even if you have not issued a read to the terminal. 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 P1, provide the name of a subroutine (must be defined as EXTERNAL) to be executed when the interrupt occurs. For P2, 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 which will 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 the user types 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 main line activities. In general, the AST subroutine should simply notify the main line 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 FORTRAN I/O using SYS$INPUT or SYS$OUTPUT (for example, by specifying UNIT=*), 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 types 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: if CTRL/Y interrupts are disabled at DCL level (SET NOCONTROL_Y) before your program is executed; if your program disables DCL CTRL/Y

interrupts with the LIB$DISABLE_CTRL Run-Time Library routine; 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 main line code. Do not attempt to connect to the terminal from AST level if you do so in your main line code.

The following program permits the terminal user to interrupt a display to see how many lines have been typed so far.

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))
! 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
```

8.3.2 Unsolicited Input

You can detect input from the terminal even if you have not issued a read 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 main line 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 three arguments:

1 The pasteboard identification number (use the value returned by SMG$CREATE_PASTEBOARD)
2 The name of an AST subprogram
3 An argument to be passed to the AST subprogram

When SMG$ENABLE_UNSOLICITED_INPUT invokes the AST subprogram it passes the subprogram two arguments: the pasteboard identification number and the argument that you specified. Typically, you write the AST subprogram to read the unsolicited input with the SMG$READ_STRING Run-Time Library routine. 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.

The following example permits the terminal user to interrupt the display of a series of arrays to either go on to the next array (by typing input beginning with an uppercase N) or to exit from the program (by typing input beginning with anything else).

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$ENABLE_UNSOLICITED_INPUT
! AST routine
EXTERNAL  AST_ROUTINE
! Create a pasteboard
STATUS = SMG$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
```

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
```

8.3.3 Type-Ahead Buffer

Normally, if the user types on the terminal before you issue a read, the input is saved in a special data structure maintained by the system called the type-ahead buffer. When you do issue a read to 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 when you issue a read by reading from the terminal with the SMG$READ_STRING Run-Time Library routine, and modifying the read operation (argument 5) with TRM$M_TM_PURGE. Clearing the type-ahead buffer has the effect of reading only what the user types on the terminal after the read is issued. Any characters in the type-ahead buffer are lost.

```
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 $IODEF). You can turn off the type-ahead buffer for further read operations with a QIO set mode operation which specifies TT$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.

8.3.4 Echo

Normally, the system writes back to the terminal any printable characters the user types on the terminal. The system also writes highlighted words in response to certain control characters; for example, the system writes EXIT if the user types 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 issue a read by reading from the terminal with the SMG$READ_STRING Run-Time Library routine, and modifying the read operation (argument 5) with TRM$M_TM_NOECHO. You can turn off echoing just for control characters 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 $IODEF). You can turn off echoing for further read operations with a QIO set mode operation which specifies TT$M_NOECHO as a basic terminal characteristic.

8.3.5 Timeout

You can restrict the user to a certain amount of time in which to respond to a read command by reading from the terminal with the SMG$READ_STRING Run-Time Library routine, and specifying argument 6. Specify the argument as the number of seconds to which the user is restricted. If the user fails to type a character in the allotted time, the error condition SS$_TIMEOUT (defined in $SSDEF) is returned. The following example restricts the user to eight seconds in which to respond to a read.

```
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 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 would have to use the SYS$SETIMR system service. The SYS$SETIMR executes an AST routine at a specified time. The specified time would be the input time limit; the AST routine would cancel any outstanding I/O on the channel assigned to the user's terminal.

8.3.6 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$READ_STRING Run-Time Library routine, and modifying the read operation (argument 5) with TRM$M_TM_CVTLOW.

```
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).

8.3.7 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 the SMG$READ_STRING Run-Time Library routine, and modifying the read operation (argument 5) with TRM$M_TM_NOFILTR.

```
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).

8.3.8 Broadcasts

You can write (broadcast) to any interactive terminal with 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))
```

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).

You can handle broadcasts to the terminal on which your program is running with the SMG$SET_BROADCAST_TRAPPING Run-Time Library routine. 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.

The 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$SET_BROADCAST_TRAPPING to identify the terminal in question. Read the contents of the broadcast with the SMG$GET_BROADCAST_MESSAGE Run-Time Library routine.

The following example 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 8.2.

```
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))
          .
          .
          .

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
```

8.4 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 FORTRAN I/O and Run-Time Library routines, reduce overhead, and permit asynchronous I/O operations. However, for ease of programming and device independence, you should generally avoid issuing QIOs.

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$QIOW system service. The I/O channel must be defined as an INTEGER*2 data type. To read from SYS$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.

8.4.1 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 completes), use SYS$QIOW. Specify the following arguments:

- I/O channel (argument 2)—Use the channel returned by SYS$ASSIGN. The argument must be a word passed by value (%VAL).

- Function code and modifiers (argument 3)—Specify the function code as IO$_READPROMPT to issue a prompt, or IO$_READVBLK to read a data record without prompting. Combine any modifiers (for example, for special formatting) with the function code using a logical .OR. operation. For example, to inhibit echoing during the read operation, specify the third argument as IO$_READVBLK .OR. IO$M_NOECHO. The symbols are defined in $IODEF. The argument must be passed by value (%VAL).
- Status block (argument 4)—Define a status block of four words: the return status, the offset of the line terminator in the input buffer, the value of the first character of the line terminator, and the size of the line terminator. The full line terminator is placed in the input buffer immediately after the input data.

  The status returned in the status block is the final status of the I/O operation; this status value is not available until the I/O operation completes. The status returned as the function value of the SYS$QIO or SYS$QIOW routine is the final status of the call to the routine; this status value is available when the routine returns control to your program. The two return values are different, equally important, and should both be checked.
- Input buffer (argument 7, or P1)—Define the input buffer as a character variable but pass it to SYS$QIOW by reference (%REF).
- Input buffer size (argument 8, or P2)—Specify the number of characters defined for the input buffer. For example, if you specify INPUT as CHARACTER*132, give INPUT_BUFF_SIZE a value of 132. The argument must be passed by value (%VAL).
- Prompt buffer (argument 11, or P5)—Define the prompt (if you are prompting) as a character literal or variable but pass it to SYS$QIOW by reference (%REF).
- Prompt buffer size (argument 12, or P6)—Specify the number of characters defined for the prompt (if you are prompting). The argument must be passed by value (%VAL).

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.)

The following example reads a line of data from the terminal and waits for the I/O to complete.

```
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 '($IODEF)'
! Status block for QIOW
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,               ! Return status
2           TERM_OFFSET,          ! Location of line terminator
2           TERMINATOR,           ! Value of terminator
2           TERM_SIZE             ! Size of terminator
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Subprograms
INTEGER*4 SYS$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$QIO system service. If the I/O is not complete, your program will pause until it is. In this manner, you can overlap processing within your program. Naturally, you must take care not to use data returned by the I/O operation before issuing SYS$SYNCH. The following FORTRAN example demonstrates an asynchronous read operation.

```
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))
! 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).

8.4.2 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). Specify the following arguments:

- I/O channel (argument 2)—Use the channel returned by SYS$ASSIGN. You must pass the argument as a word by value (%VAL).
- Function code and modifiers (argument 3)—Specify the function code as IO$_WRITEVBLK. Combine any modifiers with the function code using a logical OR operation. For example, to perform the operation without formatting the output, specify the third argument as IO$_WRITEVBLK .OR. IO$M_NOFORMAT. The symbols are defined in $IODEF. You must pass the argument by value (%VAL).

- Status block (argument 4)—Define a status block of three words and two bytes: the return status, the number of bytes written (includes any carriage control characters specified in P4 for local devices), the number of lines written (the number of line feeds transmitted), the column number at the end of transmission, and the line number at the end of transmission. The last two items are useless if you are doing any special formatting.
- Output buffer (argument 7, or P1)—Define the output buffer as a character value but pass it to SYS$QIOW by reference (%REF).
- Output buffer size (argument 8, or P2)—Specify the number of characters in the output value. You must pass the argument by value (%VAL).
- Carriage control specifier (argument 10, or P4)— Omit this argument only if you modify the function with IO$M_NOFORMAT or you do not want any carriage control action after writing the line. You must pass the argument by value (%VAL).

The following example writes a line of character data to the 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
```

```
! 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
```

8.4.3 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$GETDVIW system service to make sure the logical name is associated with a terminal, as demonstrated in the following FORTRAN example. SYS$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.

```
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
```

```
! 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
```

8.4.4 Terminal Characteristics

The *VAX/VMS I/O 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 P1. To change a characteristic:

1. Issue a QIO system service with the IO$_SENSEMODE function.
2. Set or clear the appropriate bit in the structure returned to P1.
3. Issue a QIO system service with the IO$_SETMODE function passing as P1 the structure you obtained from the sense mode operation and modified.

The following example turns off the HOSTSYNC terminal characteristic. To check that NOHOSTSYNCH has been set, issue the SHOW TERMINAL command.

```
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
! 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 types CTRL/Y), you should restore the user's environment in an exit handler. See Chapter 10 for a description of exit handlers.

8.4.5 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 of the terminator in bytes 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.

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).

9 File Input/Output

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.

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.

When determining the file attributes and order of your data file, consider how you plan to access that data. File access strategies fall into the following categories:

- Complete—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 additional memory you get using LIB$GET_VM), 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 PSECT (the LINK/MAP command produces a listing that includes the length of each PSECT). For more information on local storage, see Section 2.1.
- Record by record—If your program accesses records one after another or you cannot fit the entire file into memory, you should read one record into memory at a time.

- Discrete records—If your program processes only a few records at a time, you should read only the necessary records into memory.

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.

9.1 File Operations

To access a FORTRAN file, explicitly open the file and then perform read and /or write operations on the file. (If you omit the OPEN statement, the first READ or WRITE statement implicitly opens a file using applicable default values.) Once you have finished with the file, you can explicitly close it or allow the system to close it for you when your program terminates.

9.1.1 File Attributes

In FORTRAN, when you create or open a file, you are required to specify a logical unit number. You may also specify other attributes, such as file name, file organization, and record structure. The following subsections describe common file attributes and the specifiers (keywords) of the FORTRAN OPEN statement used to set them.

A larger set of attributes can be specified using the File Definition Language (FDL; see Section 9.7). All of the file attributes can be specified using VAX RMS in a user-open routine (see Section 9.8). Typically, you only need the FORTRAN specifiers. Use FDL only when FORTRAN specifiers are unavailable. Use a user-open routine when both FORTRAN specifiers and FDL are unavailable.

9.1.1.1 Logical Unit Number

A logical unit number is an integer value that identifies a file within a program. In FORTRAN, you associate a logical unit number with a file using the OPEN statement. Thereafter, you refer to the file by its logical unit number. To specify a logical unit number in an OPEN, READ, WRITE, DELETE, or REWRITE statement, use the UNIT specifier or specify the logical unit number as the first parameter of the statement.

If more than one person is working on a program, you should generate logical unit numbers with the Run-Time Library procedure LIB$GET_LUN. LIB$GET_LUN returns a logical unit number unique among the numbers returned by LIB$GET_LUN within the current program. This implies that

LIB$GET_LUN is only effective when used to generate all logical unit numbers within the program.

9.1.1.2 File Name

Use the FORTRAN FILE specifier to specify a file name in an OPEN statement. The file name can be any valid file or device specification, or a logical name. If you omit the file name, FORTRAN uses FOR0nn.DAT, where nn is the logical unit number. The DEFAULTFILE specifier can be used to provide a default device, directory, and file type. If you omit DEFAULTFILE, the run-time default device and directory are used with a file type of DAT.

9.1.1.3 File Organization and Access

File organization is the way in which records are arranged within a file. (The following file organizations may not be appropriate for your language, see your programming manual for more information.) FORTRAN allows three file organizations:

- Sequential—Records are arranged one after another in the order in which they are written to the file. Records can only be added to the end of the file. Records can be accessed sequentially. If you have fixed-length records, they can also be accessed directly.
- Relative—Records are arranged in fixed-length cells. The cells are numbered from 1 to n beginning with the first record in the file. Records can be placed in or deleted from any cell. Your program is responsible for keeping track of the cell number for each record. Records can be accessed sequentially or directly.
- Indexed—Records are arranged according to key fields. Each record must have at least one key, but can have more than one. The length of each key field and its position within the record are the same for every record in an indexed file. Use the value stored in a key field to identify a record. Records can be accessed sequentially or by key.

Specify the organization of a file with the ORGANIZATION specifier, which accepts the value SEQUENTIAL (default), INDEXED, or RELATIVE. If you create a relative or indexed file, you must use the RECL specifier to specify the length of the records in the file.

File access is the manner in which the records of a file are accessed. FORTRAN provides three methods of file access:

- Sequential—Records are accessed sequentially. For sequential files, the records are accessed in the order in which they were written to the file. For relative files, records are accessed according to ascending cell numbers. For indexed files, records are accessed according to ascending key values; if two keys are the same, the records are accessed in the order in which they were written to the file.

- Direct—Records are accessed directly. Specify a record by including a REC specifier in the READ or WRITE statement.
- Keyed—Records are accessed by key. Specify a record by including a KEY specifier in a READ statement. You can use sequential and keyed access on the same file (see Section 9.4.4).

Other languages may allow different methods to access files. See your language-specific programming manual for this information.

Specify file access with the ACCESS specifier, which accepts the value SEQUENTIAL (default), KEYED, DIRECT, or APPEND. APPEND indicates sequential access and, in addition, positions you at the end of the file after opening it.

The following table shows the access modes allowed for each file organization.

		Access Mode		
		Sequential	**Direct**	**Keyed**
File Organization	**Sequential**	yes	yes[1]	no
	Relative	yes	yes	no
	Indexed	yes	no	yes

[1] Only with fixed-length records

ZK-1991-84

9.1.1.4 Record Structure

Record structure determines the format of the records in a file. FORTRAN allows three record structures:

- Fixed—All records in a single file are the same length. If you specify fixed-length records, you must specify the length of the records using the RECL specifier.

- Variable—Records in a single file vary in size depending on their content. To access variable-length records in a sequentially organized file using unformatted I/O, you must specify RECORDTYPE = 'VARIABLE' when you open the file. Otherwise, FORTRAN assumes that the variable-length records are segmented records and reads the first two bytes of each record as control information.
- Segmented—Each logical record consists of one or more variable-length records in a sequentially organized file. The first two bytes of each record contain control information indicating whether the particular variable-length record is the first segment of a segmented record, the last segment, the only segment, or a middle segment.

  Segmented records are FORTRAN specific. If a file must be read by a program written in a language other than FORTRAN, do not use segmented records.

Specify the record structure of a file with the RECORDTYPE keyword of the OPEN statement, which accepts the following values: FIXED, VARIABLE, or SEGMENTED. The following table shows the record structure allowed for each file organization.

		Record Structure		
		Fixed	**Variable**	**Segmented**
File Organization	**Sequential**	yes	yes	yes[1]
	Relative	yes	yes	no
	Indexed	yes	yes	no

[1] Only with unformatted sequential access

ZK-1990-84

9.1.1.5 File I/O

Data can be read (in FORTRAN) from a record using either unformatted or formatted I/O. Unformatted I/O transfers data exactly as it is stored in memory. The data is said to be in machine-readable or binary form. Files written using unformatted I/O can be processed faster and require less storage space, but are not readily intelligible (you cannot display them with the TYPE command). You should use unformatted files only for storage of data that is written and read by programs.

Formatted I/O changes data from binary format to character format during write operations, and from character to binary format during read operations. You should use formatted I/O if you want users to be able to examine the file without using a program that knows the structure of the file. When performing formatted I/O, you must include format specifiers in your READ and WRITE statements (see *Programming in VAX FORTRAN*).

Specify a file's format using the FORM specifier, which accepts a value of FORMATTED or UNFORMATTED. If a file is opened for sequential access, file format defaults to FORMATTED; if a file is opened for direct or keyed access, file format defaults to UNFORMATTED. (To access variable-length records in a sequentially structured file using unformatted I/O, you must specify RECORDTYPE = 'VARIABLE' when you open the file.)

9.1.1.6 File Status and Disposition

Specify file status using the STATUS specifier, which accepts any one of the following values:

- OLD—Opens an existing file. If the file does not exist, FORTRAN generates an error.
- NEW—Creates a new file. If the file exists, a new version of the file is created.
- UNKNOWN—Opens an existing file if one exists; otherwise, creates a new file.
- SCRATCH—Creates a new file and deletes it when the file is closed.

By default, a file is saved when it is closed. You can use the DISPOSE specifier to indicate that you want something else done with the file. The DISPOSE specifier accepts any one of the following values: KEEP (same as SAVE; the default), DELETE, PRINT, PRINT/DELETE, SUBMIT, or SUBMIT/DELETE. A file opened as a scratch file cannot be saved, printed, or submitted. A file opened for read-only access cannot be deleted.

The DISPOSE specifier may be included in either an OPEN statement or a CLOSE statement. If specified in both an OPEN and a CLOSE statement for the same file, the disposition specified in the CLOSE statement overrides that specified in the OPEN statement.

9.1.1.7 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 Section of an FDL file (see Section 9.7).

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 (specifying READONLY does not set the protection on a file). You cannot write to a file opened with the READONLY specifier.

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.

If you are sharing a FORTRAN file, a read operation or a direct access write operation may return a status of FOR$_SPERECLOC indicating that the record you attempted to read is being used by another process. In serious programming efforts, you should include conditional code to handle this possibility. In the following FORTRAN program 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 variable 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 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.

9.1.1.8 Storage Allocation

When you open a file, you can specify the following disk storage requirements for the file:

- Initial size of the file—To specify the initial size of a file, use the INITIALSIZE specifier. If you omit INITIALSIZE, or specify it as zero, FORTRAN makes no initial allocation of storage.
- Number of blocks to be added each time a file is extended—A file is extended whenever more storage is required by that file. To specify the extend size of the file, use the EXTENDSIZE specifier. If you omit EXTENDSIZE, or specify it as zero, FORTRAN uses the system default for the device containing the file.

- Number of bytes transferred by each I/O operation—To specify the number of bytes transferred by each I/O operation, use the BLOCKSIZE specifier. By default, for sequential files FORTRAN uses the buffer size specified by the DCL command SET RMS_DEFAULT (for details, see the *VAX/VMS DCL Dictionary*). By default, for relative and indexed files FORTRAN uses the smallest number of bytes that can hold a single record.
- Number of memory buffers to be used for I/O operations—To specify the number of buffers, use the BUFFERCOUNT specifier. By default, FORTRAN uses the number of buffers specified by the DCL command SET RMS_DEFAULT. The size of each buffer is determined by the BLOCKSIZE specifier.

9.1.2 Opening Files

In FORTRAN, the OPEN statement explicitly opens a file. (For a complete description of the OPEN statement, see *Programming in VAX FORTRAN*.) Section 9.1.1 describes the options that can be specified with the OPEN statement.

9.1.2.1 Opening a New or an Existing File

To create a new file, use the OPEN statement specifying NEW as the STATUS specifier value. To open an existing file, use the OPEN statement specifying OLD as the STATUS specifier value. The following program segment creates a file.

```
INTEGER*4 LUN /1/
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
INTEGER STATUS
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create the file
OPEN (UNIT=LUN,
2     FILE=FILENAME (1:FN_SIZE),
2     DEFAULTFILE='WORKDISK:[PERSONNEL]',
2     FORM='UNFORMATTED',
2     STATUS='NEW')
```

The example solicits the name of the input file from the terminal. If the user omits the device or directory name, the defaults WORKDISK and [PERSONNEL] are used.

9.1.2.2 Opening a File of Unknown Status

To open a file which may or may not exist, use the OPEN statement specifying UNKNOWN as the STATUS specifier value. The following program segment opens a file; if the file does not exist, it is created.

```
INTEGER*4 LUN
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
INTEGER STATUS,
2       LIB$GET_LUN,
2       LIB$GET_INPUT
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get a free logical 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     FORM='UNFORMATTED',
2     STATUS='UNKNOWN')
```

A STATUS specifier value of UNKNOWN opens a file whether or not it exists. If you must take additional or alternative actions depending on whether or not the file exists, you can use the INQUIRE statement with the EXIST keyword. If the file exists, FORTRAN returns a value of .TRUE. to the variable specified with the EXIST keyword; otherwise, FORTRAN returns a value of .FALSE. (Section 9.1.5 discusses INQUIRE). The following FORTRAN example opens a scratch file if an existing file cannot be found.

```
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
LOGICAL   EXIST
INQUIRE (FILE = FILENAME (1:FN_SIZE),
2        EXIST = EXIST)
IF (EXIST) THEN
  OPEN (UNIT = LUN,
2       FILE = FILENAME (1:FN_SIZE),
2       FORM = 'UNFORMATTED',
2       STATUS = 'OLD')
ELSE
  OPEN (UNIT = LUN,
2       FILE = FILENAME (1:FN_SIZE),
2       FORM = 'UNFORMATTED',
2       STATUS = 'SCRATCH')
END IF
        .
        .
        .
```

9.1.2.3 Opening a File Across the Network

When opening files across the DECnet network, you must include the node in the file specification. The following example creates a new file on the node LIGHT on the device DISK1:.

```
OPEN (UNIT=LUN,
2     FILE='LIGHT::DISK1:[FORTRAN]INPUT.DAT',
2     FORM='UNFORMATTED',
2     STATUS='NEW')
```

If the directory or file on the remote node is protected, either the user of your program must have a proxy account that grants access to the file, or you must specify an access control string as part of the file specification.

- Proxy account—A proxy account allows a user the same rights and privileges as a specified user on the remote node. The user of your program must obtain a proxy account from the system manager of the remote node.
- Access control string—An access control string specifies the user name and password of a user on a remote node. When an access control string ("username password") is included in a file specification between the node name and the double colon delimiter, the access rights of the specified user are used to gain access to the file. The following statement opens a file on a remote node; access to the file is granted or denied depending on the access rights of the user MARX (password OHFISH).

  ```
  OPEN (UNIT=LUN,
  2     FILE= 'LIGHT"MARX OHFISH"::DISK1:[FORTRAN]INPUT.DAT',
  2     FORM='UNFORMATTED',
  2     STATUS='NEW')
  ```

Since including an access control string in your program reveals a user's password, use of proxy accounts is preferred.

9.1.3 Choosing Keywords for I/O Statements

Different types of access (sequential, direct, indexed, internal) allow different types of I/O (formatted, unformatted, list-directed, or name-list directed). In addition, most FORTRAN I/O statements (TYPE, READ, and so on) work only for certain combinations of access and I/O types, as shown in the following table.

Type of Access	Type of I/O: Formatted	Unformatted	List-Directed	Namelist-Directed
Sequential	ACCEPT PRINT READ TYPE WRITE	READ WRITE	ACCEPT PRINT READ TYPE WRITE	ACCEPT PRINT READ TYPE WRITE
Direct	READ WRITE	DELETE READ WRITE		
Indexed	READ REWRITE WRITE	DELETE READ REWRITE WRITE		
Internal	READ WRITE		READ WRITE	

ZK-1989-84

The control list of an I/O statement contains one or more specifiers (keywords) that determine the exact operation of the statement. The I/O specifiers are in either keyword form (that is, keyword = value) or nokeyword form (that is, just the value). If you use the keyword form of the specifiers, the specifiers may be in any order. If you use one or more nokeyword forms, the following rules apply.

1 The nokeyword form of the UNIT specifier must be listed first.

2 When used, the nokeyword form of the FMT, NML, or REC specifier must be listed second. If you use the nokeyword form of the FMT, NML, or REC specifier, you must also use the nokeyword form of the UNIT specifier.

The following table indicates which specifiers to use and which to omit to perform an I/O operation for a particular combination of access and I/O type. For clarity, the table uses the keyword form of each specifier.

Type of Access		Type of I/O			
		Formatted	Unformatted	List-Directed	Namelist-Directed
Sequential	Specify:	UNIT=u, FMT=f	UNIT=u	UNIT=u, FMT=*	UNIT=u, NML=nml
	Omit:	NML=nml, REC=r	NML=nml, FMT=f, FMT=*, REC=r	NML=nml, REC=r	FMT=f, REC=r
Direct	Specify:	UNIT=u, FMT=f, REC=r	UNIT=u[1], REC=r		
	Omit:	END=s, NML=nml	END=s, FMT=f, FMT=*, NML=nml		
Indexed	Specify:	UNIT=u[1], FMT=f[1], keyspec[2]	UNIT=u[1], keyspec[2]		
	Omit:	END=s, NML=nml, REC=r	END=s, FMT=f, FMT=*, NML=nml, REC=r		
Internal	Specify:	UNIT=c, FMT=f		UNIT=c, FMT=f	
	Omit:	NML=nml, REC=r		END=s, NML=nml, REC=r	

[1] The nokeyword forms of the UNIT and FMT specifiers are invalid.
[2] The term *keyspec* refers to any one of the following specifiers: KEYID, KEY, KEYEQ, KEYGT, or KEYGE.

ZK-1992-84

9.1.4 Repositioning Within a File

Use the FORTRAN REWIND statement to return to the beginning of an already open sequential file. REWIND requires only the logical unit number of the file. For a complete description of the REWIND statement, see *Programming in VAX FORTRAN.*

Typically, you use the REWIND statement to return to the beginning of a sequential file after an end-of-file is encountered; that way the file can be used later in the program without having to be closed and reopened. The following program segment uses REWIND to return to the beginning of a file after reading it.

```
INTEGER STATUS,
2       LUN1
INTEGER*2 IOSTAT
INCLUDE'($FORDEF)'
DO WHILE (STATUS .NE. FOR$_ENDDURREA)
        .
        .
        .
  READ (UNIT=LUN1,
2       IOSTAT=IOSTAT) REC_SIZE, RECORD
  IF (IOSTAT .EQ. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
  END IF
END DO
REWIND (UNIT=LUN1)
        .
        .
        .
```

9.1.5 Getting Information About a File

You can determine the characteristics of an existing file from within a program by using the FORTRAN INQUIRE statement. INQUIRE returns the state of most file attributes. In addition, INQUIRE permits you to determine whether a specific file exists, whether it is currently open, and with what logical unit number it is associated. The file you wish to inquire about can be identified by the file name or, if the file is open, by its logical unit number. For a complete description of the INQUIRE statement, see *Programming in VAX FORTRAN.*

The following example determines whether a file exists, the organization, the record type, and the record length of the file.

```
INTEGER*4 LUN /1/
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
! Returned information
LOGICAL      EXIST
CHARACTER*10 ORGANIZATION
CHARACTER*9  RECORDTYPE
INTEGER*4    RECL
INTEGER STATUS,
2       LIB$GET_INPUT
        .
        .
        .
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                        'File to inquire about: ',
2                        FN_SIZE)
! Perform an INQUIRE operation
INQUIRE (FILE         = FILENAME (1:FN_SIZE),
2        EXIST        = EXIST,
2        ORGANIZATION = ORGANIZATION,
2        RECORDTYPE   = RECORDTYPE,
2        RECL         = RECL)
        .
        .
        .
```

9.1.6 Closing a File

All files are closed automatically when a program exits. The FORTRAN CLOSE statement explicitly closes a file during program execution. Use the DISPOSE specifier to indicate what should happen to the file when it is closed; by default, a file is saved (see Section 9.1.1.6). For a complete description of the CLOSE statement, see *Programming in VAX FORTRAN*.

You should close a file explicitly in the following cases:

- If the program is going to continue after you are done with the file, close the file.
- If the file is used only by a subprogram, the subprogram should close it before returning.
- If you wish to reuse the file with different file attributes, close the file and then reopen it specifying the new attributes.
- If the program uses many files, close each file as you finish with it. Too many open files may cause a process to exceed its open file limit.

The following program segment reopens a file with write protection. Note that you can reuse the logical unit number when you reopen the file.

```
INTEGER*4     LUN /1/
INTEGER*2     FN_SIZE
CHARACTER*256 FILENAME
CHARACTER*80  INPUT_REC
! Open the file
OPEN (UNIT=LUN,
2     FILE=FILENAME (1:FN_SIZE),
2     FORM='UNFORMATTED',
2     STATUS='NEW')
        .
        .
        .
! Close the file
CLOSE (UNIT=LUN)
! Reopen the file
OPEN (UNIT=LUN,
2     FILE=FILENAME (1:FN_SIZE),
2     FORM='UNFORMATTED',
2     STATUS='OLD',
2     READONLY)
READ (UNIT=LUN) INPUT_REC
        .
        .
        .
```

9.2 Loading and Unloading a Database

To copy an entire data file from the disk to program variables and back again, either use FORTRAN I/O to read and write the data or use the SYS$CRMPSC system service to map the data. Mapping the file is faster than reading it. However, a mapped file usually uses more storage than one read using FORTRAN I/O. Using I/O, you only have to store the data that you have entered. Using SYS$CRMPSC, you have to initialize the database and store the entire structure including the parts that do not yet contain data.

9.2.1 FORTRAN I/O

When using FORTRAN I/O to access all or most of your database, perform the following operations:

1 Read all of the records from the database into program variables.

2 Process the records using the program variables.

3 Write the records to a new data file (or back to the original data file).

FORTRAN I/O can be either formatted or unformatted. Typically, you use an unformatted data file for numbers or other data accessed only by programs that understand the structure of the data file. Use a formatted data file if the data must be accessible to any general purpose display program (for example, the program invoked with the DCL command TYPE).

Each unformatted I/O operation reads or writes one record of a file. Since one or more variables (including record variables or arrays) can be written in a single I/O operation, one record of a file can contain multiple variables. The following program segment writes three records to a file: the first contains a RECORD variable, the second an INTEGER variable, and the third an array.

```
STRUCTURE /CHAR_STRING/
  INTEGER*2    LENGTH
  CHARACTER*50 NAME
END STRUCTURE
RECORD /CHAR_STRING/ REPORT_NAME
INTEGER*4 TOTAL_HOUSES
REAL*4    PERSONS_HOUSE (2048)
          .
          .
          .
WRITE (UNIT=STATS_LUN) REPORT_NAME
WRITE (UNIT=STATS_LUN) TOTAL_HOUSES
WRITE (UNIT=STATS_LUN) PERSONS_HOUSE
```

When performing I/O on arrays, reading or writing the entire array in a single I/O operation is faster than reading or writing single elements. However, if you have used only a portion of a large array, you may want to save disk space by writing only the used portion of the array to the file. In the following example, the first record of the data file STATS.SAV is an integer whose value is the number of array elements written to the file. The remaining records are the array elements, one per record, that were written.

```
                         .
                         .
                         .
! Write new STATS.SAV
IF (TOTAL_HOUSES .NE. 0) THEN
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='NEW',
2       FORM='UNFORMATTED')
  ! Unload database
  WRITE (UNIT=STATS_LUN) TOTAL_HOUSES
  WRITE (UNIT=STATS_LUN) (PERSONS_HOUSE (I), I = 1, TOTAL_HOUSES)
  WRITE (UNIT=STATS_LUN) (ADULTS_HOUSE (I), I = 1, TOTAL_HOUSES)
  WRITE (UNIT=STATS_LUN) (INCOME_HOUSE (I), I = 1, TOTAL_HOUSES)
END IF
                         .
                         .
                         .
```

The size and data type of the variable specified in a read or write statement must agree with the size of the record being read or written. Therefore, if any record in your database has a variable size, the size of the record must be included before the element or must be calculable. For example, if an unformatted file contains character data, you must have some mechanism for determining the length of the character strings. In the following example, the array NAME_HOUSE contains the last name of the major provider in each household. Each name is stored as a counted string—the value of the first byte of the string is the length of the string—of 51 characters. (Alternatively, you could store each string as a RECORD variable with two fields: an INTEGER field containing the length and a CHARACTER field containing the substring.)

```
! Declare variables to hold statistics
INTEGER TOTAL_HOUSES
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048)
CHARACTER*51 STORE_NAME(2048)
CHARACTER*50 NAME_HOUSE(2048)
INTEGER NAME_LEN(2048)
                .
                .
                .
! Open STATS.SAV file, which contains the following values:
!   o TOTAL_HOUSES (one integer)
!   o PERSONS_HOUSE (one real number times TOTAL_HOUSES)
!   o ADULTS_HOUSE (one real number times TOTAL_HOUSES)
!   o INCOME_HOUSE (one real number times TOTAL_HOUSES)
!   o STORE_NAME (one 51 character string time TOTAL_HOUSES)
OPEN (UNIT=STATS_LUN,
2     FILE='STATS.SAV',
2     STATUS='OLD',
2     FORM='UNFORMATTED',
2     IOSTAT=IOSTAT)
IF (IOSTAT .EQ. IO_OK) THEN
  ! If STATS.SAV exists, load database from STATS.SAV
  READ (UNIT=STATS_LUN) TOTAL_HOUSES
  READ (UNIT=STATS_LUN) (PERSONS_HOUSE (I), I = 1, TOTAL_HOUSES)
  READ (UNIT=STATS_LUN) (ADULTS_HOUSE (I), I = 1, TOTAL_HOUSES)
  READ (UNIT=STATS_LUN) (INCOME_HOUSE (I), I = 1, TOTAL_HOUSES)
  READ (UNIT=STATS_LUN) (STORE_NAME (I), I = 1, TOTAL_HOUSES)
  CLOSE (UNIT=STATS_LUN)
  DO I = 1, TOTAL_HOUSES
    NAME_LEN(I) = ICHAR(STORE_NAME(I)(1:1))
    NAME_HOUSE(I)(1:NAME_LEN(I)) = STORE_NAME(I)(2:NAME_LEN(I)+1)
  END DO
  ! If STATS.SAV does not exist, assume new database
ELSE
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_FILNOTFOU) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END IF
                 .
                 .
                 .
! Write new STATS.SAV
IF (TOTAL_HOUSES .NE. 0) THEN
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='NEW',
2       FORM='UNFORMATTED')
  ! Unload database
  DO I = 1, TOTAL_HOUSES
    STORE_NAME(I)(1:1) = CHAR(NAME_LEN(I))
    STORE_NAME(I)(2:NAME_LEN(I)+1) = NAME_HOUSE(I)(1:NAME_LEN(I))
  END DO
  WRITE (UNIT=STATS_LUN) TOTAL_HOUSES
  WRITE (UNIT=STATS_LUN) (PERSONS_HOUSE (I), I = 1, TOTAL_HOUSES)
  WRITE (UNIT=STATS_LUN) (ADULTS_HOUSE (I), I = 1, TOTAL_HOUSES)
  WRITE (UNIT=STATS_LUN) (INCOME_HOUSE (I), I = 1, TOTAL_HOUSES)
  WRITE (UNIT=STATS_LUN) (STORE_NAME (I), I = 1, TOTAL_HOUSES)
END IF
END
```

9.2.2 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 perform READ or WRITE statements.

Note

Files created using VAX RMS typically contain control information. Unless you are very familiar with the structure of these files, do not attempt to map one. 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 (**name** is the name of the common block):

```
PSECT_ATTR = name, PAGE
```

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.

9.2.2.1 Mapping a File

For a complete description of the SYS$CRMPSC system service, see the *VAX/VMS System Services Reference Manual*. Typically, you use only the following arguments to map a file.

- **inadr** (argument 1)—Define the first argument as an integer array of two elements. Specify the location of the first variable in the common block as the value of the first array element, 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.)
- **retadr** (argument 2)—Define the second argument as an integer array of two elements. SYS$CRMPSC returns the location of the first element mapped as the first array element and the location of the last element mapped as the second array element.

  The value returned in the first array element should be the same as the address passed in the first element of **inadr**. The value returned in the second element should be equal to or slightly more than (within 512 bytes, or one block) the value passed in the second element of **inadr**. If the first element of **retadr** is in error, you probably forgot to page align the common block containing the mapped data. If the second element of **retadr** is in error, you were probably creating a new data file and forgot to specify the INITIALSIZE keyword of the OPEN statement (see Section 9.2.2.3).

- **flags** (argument 4)—Typically, when using private sections, you specify SEC$M_WRT (indicates that the section is writable). If the file is new, also specify SEC$M_DZRO (indicates that the section should be initialized to zero; Section 9.2.2.3 discusses new sections).
- **chan** (argument 8)—You must use a user-open routine to get the channel number (see Section 9.2.2.2).

The following example 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.

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$SIGNAL (%VAL (STATUS))
INQUIRE (FILE = 'STATS.SAV',
2        EXIST = EXIST)
IF (EXIST) THEN
  ! Open STATS.SAV file
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='OLD',
2       USEROPEN = UFO_OPEN)
  MASK = SEC$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
```

9.2.2.2 User-Open Routine

When you open a file for mapping, you must specify a user-open routine to perform the following operations (Section 9.8 discusses user-open routines):

- Set the user-file open bit (FAB$V_UFO) in the FAB options mask.
- Open the file using SYS$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.)
- 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$L_STV) and must be returned in a common block.
- Return the status of the OPEN operation (SYS$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 FORTRAN I/O to access data in that file. Therefore, you should use the FORTRAN CLOSE statement to free the FORTRAN logical unit number associated with the file. The file is still open. You access the file with the channel number.

The following user-open routine is invoked by the example program in Section 9.2.2.1 if the file STATS.SAV exists. (If STATS.SAV does not exist, the user-open routine must invoke SYS$CREATE rather than SYS$OPEN.)

UFO_OPEN.FOR

```
INTEGER FUNCTION UFO_OPEN (FAB,
2                          RAB,
2                          LUN)
! Include VAX 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
! 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
```

9.2.2.3 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 9.2.2:

- Specify the size of the file—SYS$CRMPSC maps data based on the size of the file. Therefore, when creating a file that is to be mapped, you must use the INITIALSIZE specifier of the OPEN statement to create a file large enough to contain all of the expected data. Figure the size of your database as follows:
  1. Find the size of the common block (in bytes) by subtracting the location of the first variable in the common block from the location of the last variable in the common block and adding the size of the last element.
  2. Find the number of blocks in the common block by adding 511 to the size and dividing the result by 512 (512 bytes = 1 block).
- Initialize the file when you map it—FORTRAN does not initialize the blocks allocated to a file; they contain random data. When you first map the file, you should initialize the mapped area to zeros by setting the SEC$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.

9.2.2.4 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 following program segment 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$CRMPSC. For complete descriptions of the SYS$DELTVA and SYS$DASSGN system services, see the *VAX/VMS System Services Reference Manual*.

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 (argument 5) of SYS$UPDSEC to specify an event flag to be set when the update is complete and wait for the system service to complete before continuing. For a complete description of the SYS$UPDSEC system service, see the *VAX/VMS System Services Reference Manual*.

```
! 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))
                    .
                    .
                    .
```

9.3 Per-Record Processing of Entire Database

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.

9.3.1 Creating a Sequential File

To create a sequential file, use the OPEN statement specifying the following:

STATUS ='NEW', ACCESS = 'SEQUENTIAL', and ORGANIZATION = 'SEQUENTIAL'.

(ORGANIZATION may be also be specified as 'INDEXED' or 'RELATIVE'; see Section 9.4.4 for information about sequential access of indexed files.)

The following example creates 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
```

9.3.2 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.

The following program example 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.

```
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')
```

```
! 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
```

9.3.3 Sorting and Merging Sequential Files

The Sort Utility 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 from two to ten presorted input files into a single output file.

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 *VAX/VMS Sort Reference Manual*. Use the SOR$ library procedures to sort and merge files within a program. The *VAX/VMS Utility Routines Reference Manual* contains complete specifications for the procedures and their arguments.

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.

A program can perform multiple sort operations concurrently by specifying the context argument when calling the various SOR$ procedures. 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.

9.3.3.1 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 procedure. 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 SOR$ procedures. The procedures and calling sequence depend on (1) whether you are sorting or merging, and (2) which interface you use.

9.3.3.2 Sorting with the File Interface

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 *VAX/VMS Utility Routines Reference Manual*.
2. Call SOR$BEGIN_SORT to pass key information. You may also specify a number of sort options, including a user-written key comparison routine; see the *VAX/VMS Utility Routines Reference Manual*. When using the file interface, SOR$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.

The following example sorts 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$K_DTYPE_T) ! Character data
KEY_BUFFER (3) = 0                   ! Ascending sort
KEY_BUFFER (4) = 0                  ! Start at offset 0 (pos. 1)
KEY_BUFFER (5) = 20                 ! Length of the key
! Get input file name
STATUS = LIB$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
```

9.3.3.3 Sorting with the Record Interface

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 wish 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.

The following example sorts a sequential file using the file 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 may be interpreted as various fields or as a single character string field.

```
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))
! 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$SIGNAL (%VAL (STATUS))
! Release records from SORT
STATUS = SOR$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
```

9.3.3.4 Merging with the File Interface

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 *VAX/VMS Utility Routines Reference Manual*.
2. Call SOR$BEGIN_MERGE to pass key information and merge options. You may also specify a number of merge options, including a user-written key comparison routine; see the *VAX/VMS Utility Routines Reference Manual*. When using the file interface, SOR$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.

The following example merges two 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))
! 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
```

9.3.3.5 Merging with the Record Interface

To merge sequential files using the record interface:

1. Call SOR$BEGIN_MERGE to pass the key information, the longest record length, the merge order (number of input files to be merged), and a user-written input routine. The last three arguments mentioned (**lrl**, **merge_order**, and **user_input**) are required when using the record interface; the first argument (**key_buffer**) may be omitted if you specify a key comparison routine. The user input routine must determine which input file to read, read a record, and determine the record's length.
2. Call SOR$RETURN_REC once for each record that is 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: a record buffer, a file order argument (an integer passed by SORT determining which input file should be read), a record length buffer (an integer), and 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.

The following example 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 may be interpreted as various fields or as a single character string field.

```
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
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))
! 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
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
```

9.4 Processing Parts of the Database

Indexed files consist of records that contain sets of keys. Each key in a set of keys occupies the same position in its record. Each set of keys constitutes one index into the records of a file. This record structure permits

- Direct access—Access a given record by specifying the value of one of its keys.
- Sequential access—Access records sequentially according to the ascending values of a set of keys.

Keys can be either integer or character data types. An indexed file must have at least one key, known as the primary key, and may have up to 254 alternate keys.

For example, suppose that you must save the following information on customer orders:

- Order number
- Customer identification number
- Item number

You want to be able to access orders directly either by order number or customer identification number. Specify order number as the primary key because it is unique. Specify the customer identification number as the secondary key.

9.4.1 Creating an Indexed File

To create an indexed file, use an OPEN statement with the following specifiers:

- STATUS = 'NEW'
- ORGANIZATION = 'INDEXED'
- ACCESS = 'KEYED'
- RECL = n

  where **n** is the exact record size for fixed-length records and the maximum record size for variable-length records. Specify size in bytes for formatted files and in longwords for unformatted files.
- KEY = (begin:end:type,...)

where the values specified are the beginning and ending byte positions of the field, and the data type of the key (must be INTEGER*2 (2 bytes), INTEGER*4 (4 bytes), or CHARACTER.

Specify the primary key (key 0) first, the first alternate key (key 1) second, and so on. The following example shows a record definition and its matching key specifier. (A record variable is used for the record definition to ensure that the record fields are contiguous.)

Record Definition:

```
STRUCTURE /ITEM/
 INTEGER*4    ORDER_NUMBER  ! Pos 1--4  (key 0)
 CHARACTER*10 CUSTOMER_ID   ! Pos 5--14 (key 1)
 INTEGER*2    ITEM_NUMBER   ! Pos 15--16
END STRUCTURE
```

Key Specifier:

```
KEY=(1:4:INTEGER, 5:14:CHARACTER)
```

By default, duplicate values are not allowed for the primary key, that is, no two records in the file can have the same value for a primary key. Duplicate values are allowed for alternate keys. See Section 9.4.8 for more information about duplicate keys.

The following example creates an indexed file with ORDER_NUMBER as the primary key and CUSTOMER_ID as the alternate key.

```
INTEGER STATUS,
2       LUN
INTEGER*2 ORDER_SIZE,
2         ITEM_SIZE
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
CHARACTER*4 ORDER_INT
CHARACTER*2 ITEM_INT
! Define a record
STRUCTURE /ITEM/
 INTEGER*4 ORDER_NUMBER      ! Pos 1--4  (key 0)
 CHARACTER*10 CUSTOMER_ID    ! Pos 5--14 (key 1)
 INTEGER*4 CUSTOMER_LEN      ! Pos 15--16
 INTEGER*2 ITEM_NUMBER       ! Pos 19--20
END STRUCTURE
RECORD /ITEM/ TEMP
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=5,
2     KEY=(1:4:INTEGER, 5:14:CHARACTER),
2     STATUS='NEW')
                .
                .
                .
```

9.4.2 Writing to an Indexed File

A WRITE statement inserts a new record into the file and updates the indexes so that the record may be accessed by each of its keys. The record being written must include the primary key. However, you may omit an alternate key if it is at the end of a variable length record.

The following program writes an indexed file using terminal input (error checking for proper input has been omitted).

```
INTEGER STATUS,
2       LUN
INTEGER*2 ORDER_SIZE,
2         ITEM_SIZE
CHARACTER*256 FILENAME
INTEGER*2 FN_SIZE
CHARACTER*4 ORDER_INT
CHARACTER*2 ITEM_INT
! Define a record
STRUCTURE /ITEM/
 INTEGER*4 ORDER_NUMBER      ! Pos 1--4  (key 0)
 CHARACTER*10 CUSTOMER_ID    ! Pos 5--14 (key 1)
 INTEGER*4 CUSTOMER_LEN
 INTEGER*2 ITEM_NUMBER       ! Pos 15--16
END STRUCTURE
RECORD /ITEM/ TEMP
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=5,
2     KEY=(1:4:INTEGER, 5:14:CHARACTER),
2     STATUS='NEW')
! Instructions
STATUS = LIB$PUT_OUTPUT ('To exit, enter an order number of 0.')
IF (.NOT. STATUS)  CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = LIB$PUT_OUTPUT (' ')
IF (.NOT. STATUS)  CALL LIB$SIGNAL (%VAL(STATUS))
! Get the record input
STATUS = LIB$GET_INPUT (ORDER_INT,
2                        'Order Number: ',
2                        ORDER_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Convert to integer
READ (UNIT=ORDER_INT(1:ORDER_SIZE),
2     FMT='(BN,I)') TEMP.ORDER_NUMBER
DO WHILE (TEMP.ORDER_NUMBER .NE. 0)
  STATUS = LIB$GET_INPUT (TEMP.CUSTOMER_ID,
2                          'Customer ID: ',
2                          TEMP.CUSTOMER_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  STATUS = LIB$GET_INPUT (ITEM_INT,
2                          'Item Number: ',
2                          ITEM_SIZE)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Convert to integer
  READ (UNIT=ITEM_INT(1:ITEM_SIZE),
2       FMT='(BN,I)') TEMP.ITEM_NUMBER
  ! Write the record
  WRITE  (UNIT=LUN) TEMP
  ! Write blank line between records
  STATUS = LIB$PUT_OUTPUT (' ')
  ! Get the record input
  STATUS = LIB$GET_INPUT (ORDER_INT,
2                          'Order Number: ',
2                          ORDER_SIZE)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Convert to integer
  READ (UNIT=ORDER_INT(1:ORDER_SIZE),
2       FMT='(BN,I)') TEMP.ORDER_NUMBER
END DO
               .
               .
               .
```

9.4.3 Accessing a Record Directly

To access a specific record in a file, use the READ statement including the following specifiers:

- KEYID—The key of reference—specifies which key field is to be searched. KEYID takes an integer value corresponding to the key number of the key (0 for the primary key, 1 for the first alternate, and so on). For the first read operation on the file, KEYID defaults to the primary key; for subsequent read operations, KEYID defaults to the last key of reference used to access the file.
- KEY—The value of the specified key in the record you are seeking.

Instead of KEY, you could specify one of the following:

- KEYEQ—The key value of the record sought must equal that of the search value. (This specifier means the same as KEY.)
- KEYGT—The key value of the record sought must be greater than the search value.
- KEYGE—The key value of the record sought must be greater than or equal to the search value.

If no record has the specified key value, the error condition FOR$_ATTACCNON (attempt to access nonexistent record) occurs. If you attempt to access a record that is locked by another user, the error condition FOR$_SPERECLOC (specified record locked) occurs. Section 9.1.1.7 describes how to handle record lock errors.

The following program segment reads the record whose primary key is the social security number input from the terminal.

```
INTEGER STATUS,
2       LUN,
2       IOSTAT,
2       IOSTAT_OK
INTEGER*2      FN_SIZE
CHARACTER*256  FILENAME
CHARACTER*9    REQUEST_SOC
INTEGER*4      SOC_LEN
! Define a record
STRUCTURE /EMPLOYEE/
 CHARACTER*20  NAME
 CHARACTER*20  ADDRESS
 CHARACTER*15  CITY
 CHARACTER*2   STATE
 CHARACTER*6   ZIP_CODE
 CHARACTER*9   SOC_SEC_NUM
END STRUCTURE
RECORD /EMPLOYEE/ TEMP
```

```
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                        'File name: ',
2                        FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get requested social security number
STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                        'Desired SS Number: ',
2                        SOC_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
DO WHILE (SOC_LEN .NE. 9)
  ! Get requested social security number
  STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                         'Enter Full SS Number: ',
2                          SOC_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     KEY=(64:72:CHARACTER),
2     STATUS='OLD')
! Read the record
READ (UNIT=LUN,
2     KEY=REQUEST_SOC,
2     KEYID=0) TEMP
```

```
IF (IOSTAT .NE. IOSTAT_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  IF (STATUS .NE. FOR$_ATTACCNON) THEN
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END IF
DO WHILE (STATUS .EQ. FOR$_ATTACCNON)
  ! Get requested social security number
  STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                         'Desired SS Number: ',
2                         SOC_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  DO WHILE (SOC_LEN .NE. 9)
    ! Get requested social security number
    STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                           'Enter Full SS Number: ',
2                           SOC_LEN)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  END DO
  ! Read the record
  READ (UNIT=LUN,
2       KEY=REQUEST_SOC,
2       KEYID=0) TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ATTACCNON) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
  END IF
END DO
                 .
                 .
                 .
```

9.4.4 Indexed Sequential Access Method (ISAM)

You can access an indexed file sequentially, as you would a sequential file, by using a sequential READ statement. However, indexed files permit you to read selected portions of a file, rather than having to start at the beginning of the file. Proceed as follows:

- First record—Get the first record you want by specifying its key in an indexed READ statement.
- Subsequent records—Get subsequent records with sequential READ statements. Records are returned in the sequence of the key of reference used in step 1. The sequence is different depending on the key of reference.

- Last record—The last record you want to read may be the last record in the file or it may be the last record in the sequence. After each sequential read, you must check for the error condition FOR$_ENDDURREA (end-of-file), and if the read is successful, examine the record you just read to ensure that you have not passed the last record in a particular sequence.

The following program segment reads the records of a file beginning with the record greater than or equal to a user-specified social security number and ending with a record containing a social security number less than or equal to another user-specified social security number. Note that the program uses KEYGE to get the first social security number; KEYEQ might be used in cases where the exact social security number is known. Also, note that the last record is determined by checking for both end-of-file and for a social security number that exceeds the value in STOP_SOC.

```
INTEGER STATUS,
2       LUN,
2       IOSTAT,
2       IOSTAT_OK
INTEGER*2 FN_SIZE
PARAMETER (IOSTAT_OK = 0)
CHARACTER*256 FILENAME
CHARACTER*9   START_SOC,
2             STOP_SOC
INTEGER*4     SOC_LEN
INCLUDE '($FORDEF)'
! Define a record
STRUCTURE /EMPLOYEE/
 CHARACTER*20 NAME
 CHARACTER*20 ADDRESS
 CHARACTER*15 CITY
 CHARACTER*2  STATE
 CHARACTER*6  ZIP_CODE
 CHARACTER*9  SOC_SEC_NUM
END STRUCTURE
RECORD /EMPLOYEE/ TEMP
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get starting social security number
STATUS = LIB$GET_INPUT (START_SOC,
2                       'Starting SS Number: ',
2                       SOC_LEN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

```
DO WHILE (SOC_LEN .NE. 9)
  ! Get requested social security number
  STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                         'Enter Full SS Number: ',
2                         SOC_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
! Get ending social security number
STATUS = LIB$GET_INPUT (STOP_SOC,
2                       'Ending SS Number:   ')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
DO WHILE (SOC_LEN .NE. 9)
  ! Get requested social security number
  STATUS = LIB$GET_INPUT (REQUEST_SOC,
2                         'Enter Full SS Number: ',
2                         SOC_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=18,
2     KEY=(64:72:CHARACTER),
2     STATUS='OLD')
! Read the first record
READ   (UNIT=LUN,
2       KEYGE=START_SOC,
2       KEYID=0) TEMP
DO WHILE ((TEMP.SOC_SEC_NUM .LE. STOP_SOC) .AND.
2           (STATUS .NE. FOR$_ENDDURREA))
                  .
                  .
                  .
  ! Read the rest of the records
  READ (UNIT=LUN,
2       IOSTAT=IOSTAT)  TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS(,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL (STATUS))
    END IF
  END IF
END DO
                  .
                  .
                  .
```

9.4.5 Accessing a Record Using Multiple Keys

In some cases, you may want to find the record that contain two or more specified attributes. For example, in an automobile dealership inventory program. If a customer requests a car with certain attributes (a radio, a specific color, a certain number of cylinders), the dealership's inventory program would identify all the cars with the requested attributes.

To identify records with two or more attributes, you must provide each record with a unique identifier (primary key) and make each attribute an alternate key. Given a set of requested attributes, the program builds a list of qualifying records for each attribute, and then compares the lists to determine which records contain all the attributes.

The following example illustrates an automobile dealership inventory program. Each record contains a number of automobile attributes, with the stock number serving as the unique identifier. The user is allowed to select any two attributes. The program performs indexed reads for the first attribute, storing the stock numbers in an array; performs indexed reads for the second attribute, storing the stock numbers in a second array; and then compares the two arrays, printing the stock numbers of the records that contain both attributes.

A subroutine called GET_ATTRIBUTE can be written which permits the user to input the requested attributes, translate them into KEY, KEYID, and KEY_SIZE parameters, and pass them back to the main program to use in READ statements.

```
IMPLICIT NONE
INTEGER STATUS,
2       LUN,
2       KEYID,
2       KEY_SIZE,
2       I,
2       ONECNT,
2       ONE_MAX,
2       TWOCNT,
2       TWO_MAX,
2       STATUS_OK,
2       IOSTAT_OK,
2       IOSTAT
INTEGER*2 FN_SIZE
CHARACTER*256 FILENAME
CHARACTER*10 KEY
CHARACTER*8 ARRAY1(255)
CHARACTER*8 ARRAY2(255)
CHARACTER*10 FIELD
DATA ONECNT /1/
DATA TWOCNT /1/
PARAMETER   (IOSTAT_OK = 0)
PARAMETER   (STATUS_OK = 1)
```

```
INCLUDE '($FORDEF)'
INCLUDE '($RMSDEF)'
! Define a record
STRUCTURE /CAR/
 CHARACTER*8  STOCK_NUMBER     ! (key 0)
 CHARACTER*10 MODEL            ! (key 1)
 CHARACTER*3  COLOR            ! (key 2)
 CHARACTER*3  ENGINE_SIZE      ! (key 3)
 CHARACTER*2  CYLINDERS        ! (key 4)
 CHARACTER*2  RADIO            ! (key 5)
 CHARACTER*4  TRANS            ! (key 6)
END STRUCTURE
RECORD /CAR/ TEMP
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=8,
2     KEY=(1:8:CHARACTER, 9:18:CHARACTER, 19:21:CHARACTER,
2          22:24:CHARACTER,25:26:CHARACTER,27:28:CHARACTER,
2          29:32:CHARACTER),
2     STATUS='OLD')
CALL GET_ATTRIBUTE (KEYID,
2                   KEY,
2                   KEY_SIZE)
! Read the record
READ   (UNIT=LUN,
2       KEY=KEY(1:KEY_SIZE),
2       KEYID=KEYID,
2       IOSTAT=IOSTAT) TEMP
IF (IOSTAT .NE. IOSTAT_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  CALL LIB$SIGNAL  (%VAL (STATUS))
END IF
! Load stock number into array1
ARRAY1 (ONECNT) = TEMP.STOCK_NUMBER
```

```
! Set field equal to key
FIELD(1:KEY_SIZE) = KEY(1:KEY_SIZE)
DO WHILE ((FIELD(1:KEY_SIZE) .EQ. KEY(1:KEY_SIZE)) .AND.
2         (STATUS .NE. FOR$_ENDDURREA))
  ! Do a sequential read
  READ (UNIT=LUN,
2       IOSTAT=IOSTAT) TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL  (%VAL (STATUS))
    END IF
  ELSE
    ! Change the value in field, if necessary
    IF (KEYID .EQ. 1) THEN
      FIELD = TEMP.MODEL
    ELSE IF (KEYID .EQ. 2) THEN
      FIELD = TEMP.COLOR
    ELSE IF (KEYID .EQ. 3) THEN
      FIELD = TEMP.ENGINE_SIZE
    ELSE IF (KEYID .EQ. 4) THEN
      FIELD = TEMP.CYLINDERS
    ELSE IF (KEYID .EQ. 5) THEN
      FIELD = TEMP.RADIO
    ELSE IF (KEYID .EQ. 6) THEN
      FIELD = TEMP.TRANS
    END IF
    ! Check for end-of-sequence and load array1
    IF (FIELD(1:KEY_SIZE) .EQ. KEY(1:KEY_SIZE)) THEN
      ONECNT = ONECNT+1
      ARRAY1(ONECNT) = TEMP.STOCK_NUMBER
    END IF
  END IF
END DO
ONE_MAX = ONECNT
STATUS = STATUS_OK
CALL GET_ATTRIBUTE (KEYID,
2                   KEY,
2                   KEY_SIZE)
! Read the record
READ (UNIT=LUN,
2     KEY=KEY(1:KEY_SIZE),
2     KEYID=KEYID,
2     IOSTAT=IOSTAT) TEMP
IF (IOSTAT .NE. IOSTAT_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  CALL LIB$SIGNAL  (%VAL (STATUS))
END IF
! Load ARRAY2
ARRAY2(TWOCNT) = TEMP.STOCK_NUMBER
```

```
! Set field equal to key
FIELD(1:KEY_SIZE) = KEY(1:KEY_SIZE)
DO WHILE ((FIELD(1:KEY_SIZE) .EQ. KEY(1:KEY_SIZE)) .AND.
2         (STATUS .NE. FOR$_ENDDURREA))
  ! Do a sequential read
  READ (UNIT=LUN,
2       IOSTAT=IOSTAT) TEMP
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL  (%VAL (STATUS))
    END IF
  ELSE
    IF (KEYID .EQ. 1) THEN
      FIELD = TEMP.MODEL
    ELSE IF (KEYID .EQ. 2) THEN
      FIELD = TEMP.COLOR
    ELSE IF (KEYID .EQ. 3) THEN
      FIELD = TEMP.ENGINE_SIZE
    ELSE IF (KEYID .EQ. 4) THEN
      FIELD = TEMP.CYLINDERS
    ELSE IF (KEYID .EQ. 5) THEN
      FIELD = TEMP.RADIO
    ELSE IF (KEYID .EQ. 6) THEN
      FIELD = TEMP.TRANS
    END IF
    ! Check for end-of-sequence and load array
    IF (FIELD(1:KEY_SIZE) .EQ. KEY(1:KEY_SIZE)) THEN
      TWOCNT = TWOCNT+1
      ARRAY2(TWOCNT) = TEMP.STOCK_NUMBER
    END IF
  END IF
END DO
TWO_MAX = TWOCNT
STATUS = STATUS_OK
! Compare the two arrays
DO I = 1, ONE_MAX
  TWOCNT = 1
  DO WHILE ((ARRAY1(I) .NE. ARRAY2(TWOCNT)) .AND.
2           (TWOCNT .LE. TWO_MAX))
    TWOCNT = TWOCNT+1
  END DO
  IF (ARRAY1(I) .EQ. ARRAY2(TWOCNT)) THEN
    ! And type it out
    TYPE *, ARRAY1(I)
  END IF
END DO
                  .
                  .
                  .
```

9.4.6 Updating a Record in an Indexed File

To update an existing record in an indexed file, read the record, modify it, and then write it back to the file using the REWRITE statement. (A WRITE statement inserts a new record.) By default, you may modify any alternate key, but not a primary key. If you attempt to modify a key field that you are not permitted to change, the error condition FOR$IOS_INCKEYCHG (inconsistent key change) occurs. Handle this error like a duplicate key error (the error code is the same); see Section 9.4.8 for more information on handling duplicate keys.

The following example updates the record with an item number of 4 (a value of 4 in the second alternate key field). A subroutine called NEWADDRESS can be written to read a new address, city, state, and zip code from the terminal, and pass this information back to the calling program unit.

```
INTEGER STATUS,
2       LUN,
2       LIB$GET_INPUT,
2       LIB$GET_LUN
INTEGER*2     FN_SIZE
CHARACTER*256 FILENAME
! Define a record
STRUCTURE /ITEM/
 INTEGER*4    ORDER_NUMBER  ! Pos 1--4 (key 0)
 CHARACTER*20 NAME          ! Pos 5--24
 CHARACTER*20 ADDRESS       ! Pos 25--44
 CHARACTER*19 CITY          ! Pos 45--63
 CHARACTER*2  STATE         ! Pos 64--65
 CHARACTER*9  ZIP_CODE      ! Pos 66--74 (key 1)
 INTEGER*2    ITEM_NUMBER   ! Pos 75--76 (key 2)
END STRUCTURE
RECORD /ITEM/ TEMP
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='VARIABLE',
2     FORM='UNFORMATTED',
2     RECL=19,
2     KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER),
2     STATUS='OLD')
```

```
! Read the record to make it current
READ (UNIT=LUN,
2     KEY=4,
2     KEYID=2) TEMP
! Get the new address
CALL NEWADDRESS (ADDRESS,
2                CITY,
2                STATE,
2                ZIP_CODE)
! Update the record
REWRITE (UNIT=LUN) TEMP
            .
            .
            .
```

9.4.7 Deleting a Record from an Indexed File

To delete a record from an indexed file, read the record and then delete it using a DELETE statement. Deleting a record removes it from all indexes in the file. The following program segment deletes the record containing the item number 3.

```
INTEGER STATUS,
2       LUN,
2       LIB$GET_INPUT,
2       LIB$GET_LUN
INTEGER*2     FN_SIZE
CHARACTER*256 FILENAME
! Define a record
STRUCTURE /ITEM/
 INTEGER*4    ORDER_NUMBER  ! Pos 1--4 (key 0)
 CHARACTER*20 NAME          ! Pos 5--24
 CHARACTER*20 ADDRESS       ! Pos 25--44
 CHARACTER*19 CITY          ! Pos 45--63
 CHARACTER*2  STATE         ! Pos 64--65
 CHARACTER*9  ZIP_CODE      ! Pos 66--74 (key 1)
 INTEGER*2    ITEM_NUMBER   ! Pos 75--76 (key 2)
END STRUCTURE
RECORD /ITEM/ TEMP
! 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='INDEXED',
2     ACCESS='KEYED',
2     RECORDTYPE='VARIABLE',
2     FORM='UNFORMATTED',
2     RECL=19,
2     KEY=(1:4:INTEGER, 66:74:CHARACTER, 75:76:INTEGER),
2     STATUS='OLD')
! Read the record to make it current
READ (UNIT=LUN,
2     KEY=4,
2     KEYID=2) TEMP
! Get the new address
CALL NEWADDRESS (ADDRESS,
2                CITY,
2                STATE,
2                ZIP_CODE)
! Update the record
DELETE (UNIT=LUN)
            .
            .
            .
```

9.4.8 Handling Duplicate Keys

By default, FORTRAN creates files that allow duplicate alternate keys, but prohibit duplicate primary keys. If you want to create a file with key attributes other than the defaults, you must use the File Definition Language (FDL); see Section 9.7. When a file contains duplicate keys, a keyed access returns the duplicates in the order in which they were written.

In cases where duplicate keys are not allowed, the error condition FOR$IOS_INCKEYCHG (inconsistent key change) occurs. The following program segment handles this error by trapping it and prompting for a new value.

```
INTEGER STATUS,
2       IOSTAT,
2       IOSTAT_OK
INTEGER LIB$GET_INPUT,
2       LIB$GET_LUN
! Define a record
STRUCTURE /ITEM/
 INTEGER*4    ORDER_NUMBER   ! Pos 1--4 (key 0)
 CHARACTER*20 NAME           ! Pos 5--24
 CHARACTER*20 ADDRESS        ! Pos 25--44
 CHARACTER*19 CITY           ! Pos 45--63
 CHARACTER*2  STATE          ! Pos 64--65
 CHARACTER*9  ZIP_CODE       ! Pos 66--74 (key 1)
 INTEGER*2    ITEM_NUMBER    ! Pos 75--76 (key 2)
END STRUCTURE
RECORD /ITEM/ TEMP
```

```
INCLUDE '($FORIOSDEF)'
         .
         .
         .
! Write a record
WRITE  (UNIT=LUN,
2       IOSTAT=IOSTAT) TEMP
IF ((IOSTAT .NE. IOSTAT_OK) .AND.
2   (IOSTAT .NE. FOR$IOS_INCKEYCHG)) THEN
  CALL ERRSNS (,,,,STATUS)
  CALL LIB$SIGNAL (%VAL(STATUS))
END IF
! Check for duplicate error
DO WHILE (IOSTAT .EQ. FOR$IOS_INCKEYCHG)
  TYPE *, 'This order number is a duplicate'
  TYPE *, 'Enter a different value'
  READ (UNIT=*,FMT=*) TEMP.ORDER_NUMBER
  ! Write the new value
  WRITE (UNIT=LUN,
2        IOSTAT=IOSTAT) TEMP
  IF ((IOSTAT .NE. IOSTAT_OK) .AND.
2     (IOSTAT .NE. FOR$IOS_INCKEYCHG)) THEN
     CALL ERRSNS (,,,,STATUS)
     CALL LIB$SIGNAL (%VAL(STATUS))
  END IF
END DO
         .
         .
         .
```

9.5 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 *VAX/VMS Librarian Reference 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 procedures allow you to compress and expand files from within a program. (For a complete description of the DCX procedures, see the *VAX/VMS Utility Routines Reference 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.

9.5.1 Compression Procedures

Compressing a file with the DCX procedures involves the following steps (an example follows):

1. Initialize an analysis work area—Use the DCX$ANALYZE_INIT procedure to initialize a work area for analyzing the records. The first (and, typically, the only) argument passed to DCX$ANALYZE_INIT is an integer variable to contain 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 a record analyzed 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$ANALYZE_DATA procedure to pass each record in the file to an analysis work area. During analysis, the data compression facility gathers information that DCX$MAKE_MAP will use 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 FORTRAN carriage control, specify CARRIAGECONTROL = 'NONE' when you open the file to be compressed.

3. Create the compression/expansion function—Use the DCX$MAKE_MAP procedure to create the compression/expansion function. You pass DCX$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$_AGAIN, repeat steps two and three until DCX$MAKE_MAP returns a status of DCX$_NORMAL indicating that a compression/expansion function has been created.

   In the following example, 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$ANALYZE_DONE procedure to delete a work area. Identify the work area to be deleted by passing DCX$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.

You cannot use an address directly in FORTRAN (see Section 2.1.5). Therefore, use the immediate value passing mechanism to 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 9.5.2.)

6 Compress each record—Use the DCX$COMPRESS_INIT procedure 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 procedure 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 9.5.2.)

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.

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
! 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$_AGAIN))
  ! Go back to beginning of file
  REWIND (UNIT = NORM_LUN)
  ! Try map again
  STATUS = GET_MAP (NORM_LUN,
2                   CONTEXT,
2                   MAP,
2                   MAP_LEN)
END DO
IF (.NOT. STATUS) CALL LIB$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)
  ! 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
! 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
! Write map length
WRITE (UNIT = LUN) MAP_LEN
! Write map
WRITE (UNIT = LUN) MAP
END
```

9.5.2 Expansion Procedures

To expand a compressed file follow these steps:

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. Best practice is to read the length of the function from the compressed file and then invoke the library procedure LIB$GET_VM to get the necessary amount of storage for the function. LIB$GET_VM returns the address of the first byte of the storage area.

   Since you cannot use an address directly in FORTRAN, use the immediate value passing mechanism to 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$EXPAND_INIT procedure to initialize a work area for expanding the records. The first argument passed to DCX$EXPAND_INIT is an integer variable to contain a context value (see step 1 in Section 9.5.1). The second argument is the address of the compression/expansion function (use the %LOC built-in function to pass the address of the array containing the function).

3. Expand the records—Use the DCX$EXPAND_DATA procedure to expand each record.

4. Clean up the work area—Use the DCX$EXPAND_DONE procedure to delete an expansion work area. Identify the work area to be deleted by passing DCX$EXPAND_DONE a context variable.

The following example 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.

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))
```

```
! 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
```

```
! 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
```

9.6 Library Procedures

A library is a specially formatted file in which you store units of data called modules. System-defined libraries are of the following types:

- Object libraries—Each object module corresponds to one library module. An object library has a default file type of OLB; an input file to an object library has a file type of OBJ.
- Shareable image libraries—Each shareable image symbol table corresponds to one library module. A shareable image library has a default file type of OLB; an input file to a shareable image library has a file type of EXE.
- Macro libraries—Each source program corresponds to one library module. A macro library has a default file type of MLB; an input file to a macro library has a file type of MAR.
- Help libraries—A help library stores specially formatted lines of text. Every level 1 line starts a new library module. A help library has a default file type of HLB; an input file to a help library has a file type of HLP.
- Text libraries—Each text file inserted into the library corresponds to one library module. A text library has a default file type of TLB; an input file to a text library has a file type of TXT.

Modules are cataloged in the library by name. For shareable image libraries, the module names are the shareable image file names. For object and macro libraries, the module names are the names of the programs (not the names of the files containing the programs). For help libraries, the module names are the names on the level 1 lines. For text libraries, the module names are user assigned, defaulting to the names of the source files. Object and shareable image libraries also catalog the modules by the names of the global symbols defined in the modules.

Use the LIBRARY command to maintain libraries at DCL command level. For a description of libraries and how to access them at the DCL command level, see the description of the Librarian Utility in the *VAX/VMS Librarian Reference Manual*.

Use the LBR$ library procedures to maintain libraries at the programming level. The *VAX/VMS Utility Routines Reference Manual* contains complete specifications for the procedures and their arguments.

9.6.1 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$ procedures; 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 *VAX/VMS Utility Routines Reference 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 four and adding one. If you do not load the significant longwords before calling LBR$INI_CONTROL, the library may be corrupted upon creation.
3. Work with the library—Call the various LBR$ procedures 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$INIT_CONTROL, LBR$OPEN, and LBR$CLOSE for writing help text with LBR$OUTPUT_HELP. Simply invoke LBR$OUTPUT_HELP.

Certain symbols used by the LBR$ procedures 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 *VAX/VMS Utility Routines Reference 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, attempt 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. The following program unit opens, or creates and opens 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,...)
QUALIFIER 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
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$OPEN (INDEX,
2                  LIBSPEC,,
2                  '.TLB')
! If library does not exist, create it
IF (STATUS .EQ. %LOC (RMS$_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
```

9.6.2 Adding Modules

Use the following procedures 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's 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$LOOKUP_KEY in step 1 above (you should not receive an error message) in one variable and the new RFA returned by the first call to LBR$PUT_RECORD (step 2) in another variable. On step 4, invoke LBR$REPLACE_KEY instead of LBR$INSERT_KEY passing the old RFA as the third argument and the new RFA as the fourth argument.

The following subroutine solicits module names and text from SYS$INPUT, and adds 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$_KEYNOTFND, ! Key not found
2        DOLIB_MODEX      ! Module already exists
! Get first module name
STATUS = LIB$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
  ! 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
```

9.6.3 Deleting Modules

Use the following procedures 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 following subroutine gets module names from the command line and deletes the specified 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)
```

```
! 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
```

9.6.4 Extracting Modules

Use the following procedures 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 following subroutine gets module names from the command line, extracts the contents of the modules, and writes the contents to SYS$OUTPUT.

```
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$SIGNAL (%VAL (STATUS))
      ! Get another record
      TEXTLINE (1:255) = ' '
      STATUS = LBR$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
```

9.6.5 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 form 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 following subroutine associates additional keys (which the routine calls aliases) with modules and stores these keys in index 2.

```
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
! 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 by (1) looking up the module (LBR$LOOKUP_KEY) using one of the keys, and (2) searching (LBR$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 following subroutine prints 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.

```
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))
! 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
```

9.6.6 Accessing Module Headers

You can store user information in the header of each module 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 (index value returned by LBR$INI_CONTROL), second (RFA returned by LBR$LOOKUP_KEY), and fifth (character string containing the user data) arguments.

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.

The following example displays the user data portion of module headers on SYS$OUTPUT and applies updates from SYS$INPUT.

```
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$LOOKUP_KEY (INDEX,
2                          MODNAME (1:MODNAME_LEN),
2                          TXTRFA)
  ! Get header if module exists
  IF (STATUS) THEN
    STATUS = LBR$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
```

9.6.7 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 *VAX/VMS Utility Routines Reference Manual*) identify the significant longwords of the array by their byte offsets into the array. Convert these values to subscripts by dividing by four and adding one.

The following example reads the library header and displays some information from it.

```
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
```

9.6.8 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. The following program solicits keywords from SYS$INPUT and displays the text associated with those keywords on SYS$OUTPUT, inhibiting the prompting facility.

```
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$SIGNAL (%VAL (STATUS))
IF (LIBSPEC_LEN .EQ. 0) THEN
  LIBSPEC = 'HELPLIB'
  LIBSPEC_LEN = 7
END IF
! Set flags for no prompting
FLAGS = %LOC (HLP$_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
```

9.6.9 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 will be 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 will contain the name of the index entry. Declare the second argument as an integer array of two elements.

The following example 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).

```
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
```

9.7 File Definition Language

The File Definition Language (FDL) commands and procedures 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 and/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 a FORTRAN OPEN statement. Instead, use FDL to create your data file, set the desired file characteristics, close the file, and then use a FORTRAN OPEN 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).

9.7.1 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, best practice is to use the FDL editor or create the FDL file from an existing data file.

9.7.1.1 Using the FDL Editor

To invoke the FDL editor, use the EDIT/FDL command. Use the editor interactively to create new FDL files or 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 9.7.2.2 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 FORTRAN, 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 9.7.2.2). 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.

9.7.1.2 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 library 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 VAX 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 *VAX/VMS DCL Dictionary*.

The following program segment creates an FDL file INCOME.FDL, which describes the VAX 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$GENERATE can be invoked only from within a user-open routine (Section 9.8 describes user-open routines). The *VAX/VMS Utility Routines Reference Manual*) contains complete specifications for FDL$GENERATE.

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$GENERATE
MASK = IBSET (MASK, %LOC(FDL$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
```

9.7.2 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.

9.7.2.1 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 FDL Utility in the *VAX/VMS File Definition Language Reference 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$CREATE; this argument is useful for debugging an FDL file. The *VAX/VMS Utility Routines Reference Manual*) contains complete specifications for FDL$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))
                      .
                      .
                      .
```

9.7.2.2 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 VAX RMS and the internal structure of the file, 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, will 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
   ```

9.8 User-Open Routines

A user-open routine allows you direct access to the FAB and RAB (the RMS structures that define file characteristics). Use a user-open routine to specify file characteristics otherwise unavailable from FORTRAN.

When you specify a user-open routine, you open the file rather than allowing FORTRAN to open the file for you. Before passing the FAB and RAB to your user-open routine, FORTRAN sets any file characteristics specified by keywords in the OPEN statement and the FORTRAN defaults. Your user-open routine should not set or modify file characteristics that can be set by FORTRAN keywords because FORTRAN may not be aware that you have set the characteristics and may not perform as expected.

Note

The FORTRAN system definition library, SYS$LIBRARY:FORSYSDEF.TLB, includes record structure definitions for most VAX RMS structures, providing direct access to VAX RMS. However, correct use of VAX RMS structures

requires a knowledge of RMS internals beyond the scope of this manual; therefore, this documentation discusses only the FAB and RAB structures (which are accessible from a user-open routine) and the following VAX RMS routines: SYS$CONNECT, SYS$CREATE, and SYS$OPEN. For more information about RMS, see the *VAX Record Management Services Reference Manual*.

9.8.1 USEROPEN Specifier

To open a file with a user-open routine, include the USEROPEN specifier in the 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 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)
        .
        .
        .
```

9.8.2 Writing a 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 at least perform the following operations. In addition, before opening the file, a user-open routine usually adjusts one or more fields in the FAB and/or the RAB.

- 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 VAX 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
```

9.8.3 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:

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 VAX 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
```

10 Run-Time Errors

Run-time errors are events, usually errors, detected by hardware or software that alter normal program execution. Examples of run-time errors are:

- System errors—For example, specifying an invalid argument to a system-defined procedure.
- Language-specific errors—For example, in 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, the VAX/VMS operating system either returns a condition code identifying the error to your program or signals the condition code (Section 10.1.3 describes signaling). If the VAX/VMS operating system signals the condition code, an error message is typically displayed, and program execution continues or terminates depending on the severity of the error. If the VAX/VMS operating system returns the condition code to your program, your program should test the condition code and respond accordingly.

Both an error message and its associated condition code identify an error by the name of the facility that generated the error and an abbreviation of the message text. (See 1.1.4 for a list of the commonly used facility abbreviations.) 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 *VAX/VMS System Routines Reference Volume* include a lists of the condition codes that may be returned by the routine.

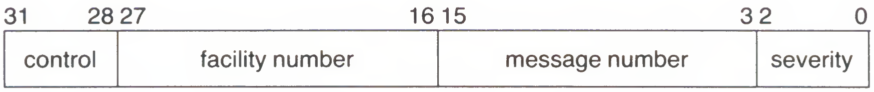
10.1 General Error Handling

When unexpected errors occur, your program should display a message that identifies the error, and 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.

10.1.1 Condition Code and Message

Error conditions are identified by integer values called condition codes. The VAX/VMS operating system defines condition codes to identify errors that may occur during execution of system-defined procedures. You can define condition codes for errors that may occur in your programs (see Section 10.2 for more information).

From a condition code you can determine whether or not any error has occurred, which particular error has occurred, and the severity of the error. A condition code contains the following fields:



ZK-2049-84

- 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 29 through 31 are reserved for DIGITAL.

Code (Symbolic Name)	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

Code (Symbolic Name)	Severity	Response
3 (STS$K_INFO)	Information	Execution continues, informational message displayed
4 (STS$K_SEVERE)	Severe error	Execution terminates, no output
5		Reserved for DIGITAL
6		Reserved for DIGITAL
7		Reserved for DIGITAL

10.1.2 Return Status Convention

Most system-defined procedures are functions of INTEGER*4 data type 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. To access a procedure's return status, you must declare the procedure name as an INTEGER*4 data type and invoke the procedure as a function. Each procedure description in the *VAX/VMS System Routines Reference Volume* lists the condition codes that may be returned by that procedure. For example, the Run-Time Library procedure LIB$SET_CURSOR returns one of three condition codes defined by the symbols SS$_NORMAL (success), LIB$_INVARG (invalid argument), or LIB$_INVSCRPOS (invalid screen position).

10.1.2.1 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$DATE_TIME, checks the returned condition code (returned in the variable STATUS), and, if an error occurred, signals the condition code by calling the Run-Time Library procedure LIB$SIGNAL (Section 10.1.3 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 (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
         .
         .
         .
```

10.1.2.2 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 may instead return a more specific condition code that indicates the privilege or quota in question. The following tables list the specific privilege errors (first table) and quota errors (second table).

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
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

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 may 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 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
```

10.1.3 Signaling Mechanism

Signaling a condition code causes the VAX/VMS operating system to pass control to a special subprogram called a condition handler. The VAX/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 10.1.3.1).

You can signal a condition code by invoking the Run-Time Library procedure LIB$SIGNAL and passing the condition code as the first argument. In FORTRAN, use the by-value passing mechanism (%VAL) to pass the condition code. (The *VAX/VMS Run-Time Library Routines Reference Manual* contains the complete specifications for LIB$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 may signal the appropriate condition code rather than returning 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.

10.1.3.1 Default Condition Handling

The VAX/VMS operating system 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.

10.1.3.2 Changing a Signal to a Return Status

If you expect a particular condition code to be signaled, you can prevent the VAX/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 10.4.

To change a signal to a return status, you must put any code that might signal a condition code into an INTEGER 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 in an EXTERNAL statement.
- 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$SIG_TO_RET changes the signal to the return status of the INTERCEPT_SIGNAL function and returns control to the program unit that invoked INTERCEPT_SIGNAL. (If GET_1_STAT has a condition handler established, the VAX/VMS operating system invokes that handler before invoking LIB$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 10.1.2) to determine which condition code, if any, was signaled during execution of INTERCEPT_SIGNAL.

10.2 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

10.2.1 Creating the Source File

A message source file contains definition statements and directives. The following source message file defines the error messages generated by the FORTRAN 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 *VAX/VMS Message Reference Manual*.

10.2.1.1 Specifying 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. Best practice is to have one .FACILITY directive per message file.

10.2.1.2 Specifying Severity

Use the .SEVERITY directive and one of the following keywords to specify the severity of one or more condition codes.

SUCCESS
WARNING
INFORMATIONAL
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
```

10.2.1.3 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 one 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"
```

10.2.1.4 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 *VAX/VMS Message Reference 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 10.2.3).

10.2.2 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. To specify an object file name different than the source file name, use the /OBJECT qualifier of the MESSAGE command. To specify an object module name different than the source file name, use the .TITLE directive in the message source file.

```
$ MESSAGE INCMSG
```

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 module, include the message object module in the program's object library. For example, to include the message module in INCOME's object library, specify

```
$ LIBRARY INCOME.OLB INCMSG.OBJ
```

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
```

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
```

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 *VAX/VMS Message Reference Manual.*

10.2.3 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 a FORTRAN 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.

10.2.3.1 Signaling with Global Symbols

To signal a user-defined condition code using a global symbol, declare the appropriate condition code in an EXTERNAL statement in the definition section of the program unit, and then invoke the Run-Time Library procedure LIB$SIGNAL to signal the condition code. Use the built-in function %LOC to reference a symbol declared in an EXTERNAL statement. 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
```

10.2.3.2 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.

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 an INTEGER*4 variable and use a PARAMETER 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 with the same name as the source file and a file type of FOR.

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 Run-Time Library procedure 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. When using local symbols, you omit %LOC; however, you must still use %VAL.

```
! 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
```

10.2.3.3 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. Use the built-in function %VAL to pass the number of FAO arguments and any numeric FAO arguments. 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

```
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

```
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 10.2.3.2.

10.3 FORTRAN I/O Errors

By default, if an error occurs during execution of a FORTRAN I/O statement, FORTRAN signals the appropriate condition code. If you are expecting a particular I/O error and want to intercept it before FORTRAN signals, include the IOSTAT specifier in the I/O statement. When IOSTAT is specified, FORTRAN returns a FORTRAN error code in the IOSTAT variable rather than signaling the appropriate condition code.

Specifying either the END or ERR specifier in an I/O statement also prevents FORTRAN from signaling. Both END and ERR circumvent normal execution sequence by transferring control to a label in your program; however, they are useful for exiting from the loops generally used for I/O. For easier code maintenance, transfer control to the statement immediately following the loop or to the statement at the beginning of the loop; otherwise, include a comment that directs the reader of your code to the statement receiving control.

10.3.1 Unexpected FORTRAN I/O Errors

If you are not expecting an I/O statement to generate an error, do not include the IOSTAT specifier. The following READ operation is expected to complete successfully.

```
! Format for READ
CHARACTER*(*) INTEGER_FMT
PARAMETER (INTEGER_FMT = '(I4)')
READ (UNIT = *,
2     FMT = INTEGER_FMT)
```

If an error occurs during the read operation, the traceback or catchall handler displays an error message (like the following) and continues or terminates program execution, depending on the severity of the error.

```
%FOR-F-INPCONERR, input conversion error
   unit 0 file WORKDISK:[ACCOUNTS]DOGS83.DAT
   user PC 00000628B
-FOR-F-INVTEXREC, invalid text is ''
```

10.3.2 Expected FORTRAN I/O Errors

To handle an expected error, include the IOSTAT specifier in the I/O statement. If the statement executes successfully, the IOSTAT variable contains a value of 0. If the statement fails, the IOSTAT variable contains a positive integer value that identifies the FORTRAN error that occurred. If the statement encounters an end-of-file condition, the IOSTAT variable contains a negative integer value.

If IOSTAT contains a positive integer, you check the value against the value of the expected error. If the error that occurred is the one you expected, you take whatever measures are appropriate; otherwise, you signal the error. When you signal a FORTRAN error, the message identifies the error, but generally includes garbled text as well. Most FORTRAN errors include FAO arguments used to provide information about the element that caused the error; the garbled text appears because you signal the error without supplying parameters for those FAO arguments. When FORTRAN signals an error, the message text is clear since the FAO parameters are automatically included.

To signal a FORTRAN error, you must first use the intrinsic subprogram ERRSNS to translate the FORTRAN error code (returned in the IOSTAT variable) to the matching VAX/VMS condition code. You cannot use LIB$SIGNAL to signal the FORTRAN error code. The FORTRAN error codes are defined in the module $FORIOSDEF. The matching VAX/VMS condition codes are defined in the module $FORDEF.

The intrinsic subprogram ERRSNS accepts five optional arguments: the fifth argument returns the VAX/VMS condition code. If the read operation in the following example executes successfully, program execution continues. If the read operation fails, ERRSNS is invoked to return the VAX/VMS condition code. If the error was an input conversion error (VAX/VMS error code FOR$_INPCONERR), the program takes special action; otherwise, and the error is signaled.

```
! Format for READ
CHARACTER*(*) INTEGER_FMT
PARAMETER     (INTEGER_FMT = '(I4)')
INTEGER*4     STAT
! Status variable and values
INTEGER STATUS,
2       IO_STAT,
2       IO_OK
PARAMETER (IO_OK = 0)
INCLUDE  '($FORDEF)'
              .
              .
              .
! Read one integer
READ (UNIT=*,
2     FMT=INTEGER_FMT,
2     IOSTAT=IO_STAT) STAT
```

```
IF (IO_STAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
  ! Check for conversion error
  IF (STATUS .EQ. FOR$_INPCONERR) THEN
                     .
                     .
                     .
  ELSE
    ! Unexpected error
    CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
END IF
```

10.4 Condition Handlers

When a program signals a condition code, the VAX/VMS operating system searches for a condition handler, invokes the first handler it finds, passing the handler information about the condition code and the state of the program when the condition code was signaled. If the handler resignals, the VAX/VMS operating system searches for another handler; otherwise, the search for a condition handler ends.

The VAX/VMS operating system searches for condition handlers in the following sequence:

- 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 VAX/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 canceled or 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.

- 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.

- 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 VAX/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 VAX/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.
- 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 VAX/VMS operating system finds and invokes the traceback handler (see Section 10.1.3.1).
- Catchall handler—If you do not establish any condition handlers and link your program with the /NOTRACEBACK qualifier of the LINK command, the VAX/VMS operating system finds and invokes the catchall handler (see Section 10.1.3.1).
- 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 Run-Time Library procedure 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 10.1.3.2 describes the system-defined handler LIB$SIG_TO_RET that allows you to treat signals as return values; Section 10.4.5 describes other useful system-defined handlers for arithmetic errors.)
- Modify part of a condition code, such as the severity (see Section 10.4.4 for more information). If you want to change the severity of any condition code to a severe error, you can use the run-time library procedure LIB$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 10.4.4 for more information).

10.4.1 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.

```
CALL LIB$ESTABLISH (OLD_HANDLER)
```

10.4.2 Writing a Condition Handler

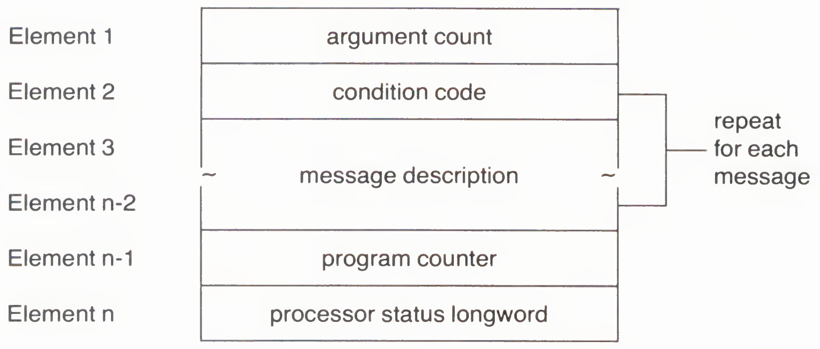
You must write your condition handler as an INTEGER function that accepts two integer arrays as dummy arguments.

10.4.2.1 Dummy Arguments

The VAX/VMS operating system passes a condition handler two arrays. Any condition handler that you write should declare two dummy arguments as variable-length arrays. For example:

```
INTEGER*4 FUNCTION HANDLER (SIGARGS,
2                           MECHARGS)
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
                        .
                        .
                        .
```

The first dummy argument, the signal array, describes the signaled condition code that indicate which error occurred and the state of the process when the condition code was signaled.

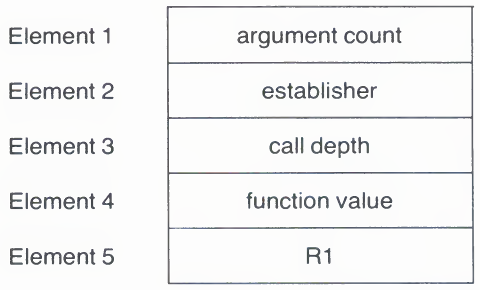


ZK-2050-84

- 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$PUTMSG description in the *VAX/VMS System Services Reference Manual.*
- Program counter (PC)—If the error that caused the signal was a fault (occurring during the instruction's execution), the PC contains the address of the instruction that signaled the condition code. If the error that caused the signal was a trap (occurring at the end of the instruction), the PC contains the address of the instruction following the one that signaled the condition code. The error generated by LIB$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.

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.



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 VAX/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. (In FORTRAN, when you invoke a function, the R0 register contains the function value.)

10.4.2.2 Checking the Condition Code

A condition handler is usually written in anticipation of a particular set of condition codes. Since a handler will be 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.

To compare the signaled condition code to a list of expected condition codes, use the Run-Time Library procedure 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 FORTRAN I/O errors. If it is not, the condition handler resignals the condition code. Note that when a FORTRAN I/O error is signaled, the signal array describes the VAX/VMS condition code, not the 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
```

10.4.2.3 Exiting

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$_CONTINUE, the handler returns control to the program at the statement that signaled the condition (fault) or the statement following the one that signaled the condition (trap). The Run-Time Library routine LIB$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 handler displays a message (Section 10.4.4.2) 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 VAX/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 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$UNWIND system service, the handler can return control to any point in the program unit that incurred the exception, the program unit that invoked the program unit that incurred the exception, and so on back to the program unit that established the handler. The remainder of this section discusses SYS$UNWIND.

Since correctly invoking SYS$UNWIND requires a knowledge of VMS internals that is 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.

- Establisher—To return control to the program unit that established the handler, invoke SYS$UNWIND passing the call depth (third element of the mechanism array) as the first argument and no second argument.

  ```
  ! Declare dummy arguments
  INTEGER*4 SIGARGS(*),
  2         MECHARGS(*)
              .
              .
              .
  CALL SYS$UNWIND (MECHARGS(3),)
  ```

- Program unit that invoked the establisher—To return control to the caller of the program unit that established the handler, invoke SYS$UNWIND passing no 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 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 no action whatsoever.

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 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
```

10.4.3 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 handler. You must set a breakpoint in the handler and wait for the debugger to invoke the handler.

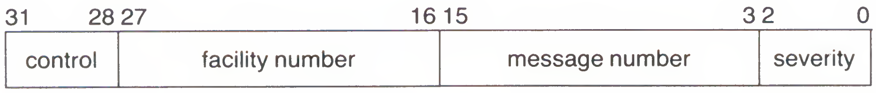
Typically, to trace execution of a condition handler, you set breakpoints at the statement in your program that should signal the condition code, at the statement following the one that should signal, and at the first executable statement in your condition handler. Chapter 5 describes how to use the debugger.

10.4.4 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.

10.4.4.1 Modifying Condition Codes

A condition code contains the following information:



ZK-2052-84

To modify a condition code, use the FORTRAN intrinsic subroutine MVBITS. MVBITS allows you to copy a series of bits from one longword to another longword. For example, the following statement copies the first three bits (bits 0 through 2) of STS$K_INFO to the first three bits of the signaled condition code, which is in the second element of the signal array named SIGARGS. As shown in the table in Section 10.1.1, STS$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 have modified the condition code, you can resignal the condition code and let the default 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 handler will display the message a second time.

In the following example, the condition handler checks to be sure 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
```

10.4.4.2 Displaying Messages

The VAX/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 VAX/VMS operating system passes to the condition handler as the first argument of the SYS$PUTMSG system service. The signal array contains the condition code, the number of required FAO arguments for each condition code, and the FAO arguments. The *VAX/VMS System Services Reference Manual* contains complete specifications for SYS$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 two from the value in the first element. Before exiting from the condition handler restore the last two elements of the array by adding two 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
```

10.4.4.3 Chaining Messages

A handler may 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 VAX/VMS operating system will signal 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 one.) 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 Run-Time Library procedure 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 two and three.

```
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
```

10.4.4.4 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. 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.

To keep a running log of the messages displayed by your program (that is, a log that is resumed each time your program is invoked), 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 *VAX/VMS System Services Reference Manual* contains complete specifications for SYS$PUTMSG.)

For example, to keep a running log of messages, you might have your main program open a file for the error log, perhaps 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$PUTMSG writes the message to the log file, the handler resignals to continue program execution. (The condition handler uses LIB$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
```

10.4.5 System-Defined Arithmetic Condition Handlers

The VAX/VMS operating system provides three arithmetic condition handlers:

- LIB$DEC_OVER—Enables/disables the signaling of a decimal overflow. By default, signaling is disabled.
- LIB$FLT_UNDER—Enables/disables the signaling of a floating-point underflow. By default, signaling is disabled.
- LIB$INT_OVER—Enables/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.
- When you compile your program with the FORTRAN command, specify the OVERFLOW (LIB$INT_OVER) option of the /CHECK qualifier to enable signaling of integer overflow or specify the UNDERFLOW (LIB$FLT_UNDER) option of the /CHECK qualfier to enable signaling of floating-point underflow. You cannot use the /CHECK qualifier to enable signaling of decimal overflow (LIB$DEC_OVER).

10.5 Exit Handlers

When an image exits, the VAX/VMS operating system performs the following operations:

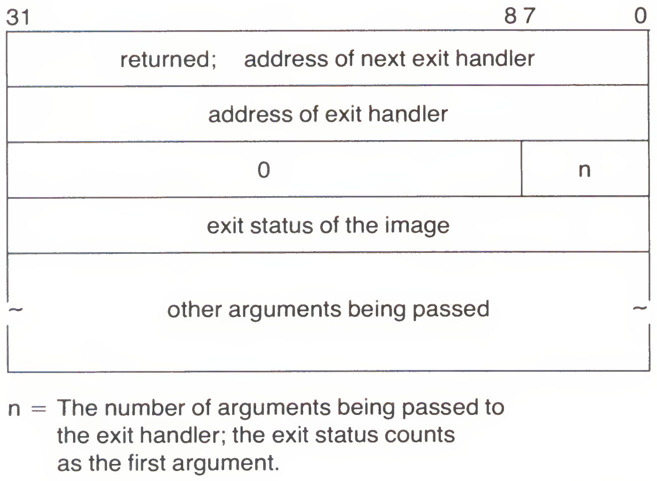
- Invokes any user-defined exit handlers.
- Invokes the system-defined default exit handler, which closes any files that were left open by the program or user-defined exit handlers.
- Executes a number of cleanup operations collectively known as image run-down. The following list contains some of these 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 VAX/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.

10.5.1 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.

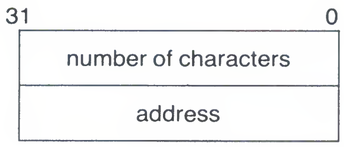


31
8 7
0
returned; address of next exit handler
address of exit handler
0
n
exit status of the image
other arguments being passed
n = The number of arguments being passed to the exit handler; the exit status counts as the first argument.
ZK-2053-84

The first longword of the structure contains the address of the next handler. The VAX/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 VAX/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 the %LOC function 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 form



ZK-2054-84

Use the %LOC function 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. (Section 1.5.8 describes how to create a variable-length data structure using a record.)

```
                 .
                 .
                 .
! 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.

10.5.2 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 Run-Time Library procedures. Trying to access the terminal from an exit handler using FORTRAN I/O may cause a redundant I/O error.

10.5.3 Debugging an Exit Handler

To debug an exit handler, you must set a breakpoint in the handler and wait for the VAX/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 has begun 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.

Index

A

Absolute time • 6-43
Access
  direct • 9-4
  keyed • 9-4
  sequential • 9-4
  specifiers • 9-11
Actual argument • 1-8
Add
  command to DCL table • 7-12
  command to process table • 7-11
  command to user table • 7-12
  module to object library • 4-8
Addition • 6-7
Address
  virtual memory • 4-18
Aligning data • 9-19
ALWAYS keyword
  GSMATCH option • 4-30
ANALYZE/RMS_FILE command • 9-94
.AND. logical operator • 6-11
Apostrophe (')
  character constant • 6-13
Argument
  actual • 1-8
  buffer address • 1-33
  byte • 1-24
  common block • 1-10
  dummy • 1-8
  list • 1-8
  longword • 1-25
  mask • 1-26, 1-30
  passing • 1-8, 1-10
  passing mechanisms • 1-22
  quadword • 1-28
  system-defined procedure • 1-20
  variable-length structure • 1-31
  word • 1-24
Arithmetic error • 6-10
Arithmetic operation • 6-7
Arithmetic trap • 4-5
Array • 4-37
  adjustable • 1-18
  assumed-size • 1-18
  bounds • 6-27
  character • 6-24
  EQUIVALENCE elements • 2-7
  I/O • 9-17
  initializing elements • 2-5
  multidimensional • 6-28
  numeric • 6-23
    defining • 6-23
    referencing • 6-24
  passing • 1-17
  processing with implied DO loop • 8-13
  referencing the full array • 6-26
  storage • 6-27
Assignment statement • 6-3
  record • 6-37
AST (asynchronous system trap) • 3-24
  delivery • 3-25
  execution • 3-24
  writing • 3-24
Asynchronous input/output • 8-58
Asynchronous read • 8-59
Asynchronous system trap
  see AST
Attribute
  video • 8-18

B

Background color
  screen • 8-17
Bit
  manipulation • 6-19
Blink • 8-18
BLOCK DATA statement • 1-12
Bold • 8-18
Border
  virtual display • 8-19

Bounds array • 6-27
/BRIEF qualifier
  LINK/MAP command • 4-40
Broadcast • 8-53
Buffered input/output operation • 6-40
Byte
  in longword • 1-25
  in quadword • 1-28
  passing • 1-24
BYTE data type • 6-4

C

CALL command • 5-17
CALL statement
  array processing • 6-26
CANCEL DISPLAY command • 5-11
Carriage control • 8-14
Catchall handler • 10-7
Channel
  input/output • 8-55, 8-59
Character
  array • 6-24
  constant • 6-13, 6-16
  conversion • 6-21
  data • 6-12
  nonprintable • 6-15
  string • 6-12
Characterconversion
  from number • 6-22
Characteristics
  terminal • 8-62
Character string
  assumed-size • 6-12
  concatenating • 6-14
  counted • 6-15
  defining • 6-12
  fixed-length • 6-12
  padding • 6-14
  passed-length • 1-16
  passing • 1-16
  referencing • 6-13
  size • 6-13
  truncating • 6-14
CHAR intrinsic function • 6-15
/CHECK=BOUNDS qualifier
  FORTRAN command • 6-27
/CHECK=NOOVERFLOW qualifier
  FORTRAN command • 6-10
/CHECK=NOUNDERFLOW qualifier
  FORTRAN command • 6-10
Checkin/checkout
  system program units • 4-15
/CHECK qualifier
  FORTRAN command • 4-5
CLI$_ABSENT • 7-15, 7-19
CLI$_COMMA • 7-18
CLI$_CONCAT • 7-19
CLI$_DEFAULTED • 7-15
CLI$_LOCNEG • 7-20
CLI$_LOCPRES • 7-20
CLI$_NEGATED • 7-15
CLI$_PRESENT • 7-15
CLI$DCL_PARSE • 7-23
CLI$DISPATCH • 7-26
CLI$GET_VALUE
CLI$PRESENT
CLOSE statement • 9-15
CLUSTER option • 4-26
Command
  parsing
  deleting • 7-14
  description • 7-1
  image • 7-1
  input • 7-1
  line • 7-22
  name • 7-1
  verb • 7-22
Command procedure
  debugger • 5-35
  system building • 4-11
Command syntax
  changing • 7-9
  defining • 7-1
Command table
  create • 7-11
  DCL • 7-12
  modify • 7-11
  user-defined • 7-12

Comment
  formatting source program • 4-3
Common block • 1-10, 2-8, 4-19
  align • 9-19
  BLOCK DATA statement • 1-12
  contents • 1-10
  initializing • 1-12
  installing • 3-43
  interprocess • 3-43
  passing • 1-31
  per-process • 3-50
  uses • 1-10
Common event flag • 3-18
Common event flag cluster
  permanent • 3-22
  temporary • 3-22
COMMON statement • 1-10, 2-8
  array processing • 6-26
  record processing • 6-37
Compile
  development system • 4-5
  listing • 4-33
  production system • 4-6
  program • 4-4
COMPLEX*16 data type • 6-6
COMPLEX*8 data type • 6-6
COMPLEX data type • 6-6
Complex number • 6-6
  constant • 6-6
Composed input • 8-37
Conditional execution • 2-14
Conditional logic
  block IF • 2-15
  exclusive conditional • 2-16
  inclusive IF • 2-19
  multiple conditions • 2-14
  nesting • 2-22
  single condition • 2-14, 2-15
Condition code • 10-2
  chaining • 10-30
  defining • 10-9
  FORTRAN error code • 10-17
  modifying • 10-28
  signaling • 10-6
  SS$_EXQUOTA • 10-4
Condition code (cont'd.)
  SS$_NOPRIV • 10-4
Condition code and message • 10-2
Condition handler
  arithmetic • 10-34
  condition code • 10-24
  debugging • 10-27
  establishing • 10-21
  exiting • 10-24
  mechanism array • 10-22
  signal array • 10-21
  use of • 10-19, 10-28
  writing • 10-21
Condition handling
  arithmetic • 10-34
  condition handler • 10-21
  default • 10-6
  FORTRAN I/O errors • 10-16
  handlers • 10-18
  last-chance exception vectors • 10-19
  primary exception vectors • 10-18
  resignaling • 10-25
  returning to other locations • 10-25
  return status • 10-3
  secondary exception vectors • 10-18
  signal • 10-6
  unwinding • 10-25
Constant • 6-2
  character • 6-13, 6-16
  complex number • 6-6
  hexadecimal • 6-17
  Hollerith • 6-19
  integer • 6-4
  logical • 6-11
  octal • 6-18
  real number • 6-5
Continuation line
  source program • 4-2
Control action
  inhibit • 8-52
Control character
  CTRL/Y • 5-3
Control information • 1-13
  flag • 1-13
  global symbols • 1-13
  indicative value • 1-15

Control information (cont'd.)
  mask • 1-14
Conversion
  automatic • 6-22
  character • 6-21
  data • 6-20
  formatted • 6-21
  input data • 8-7
  lowercase to uppercase • 8-52
  numeric • 6-20
CONVERT/FDL command • 9-96
Counted string • 6-15
CPU time • 6-40
Create
  object library • 4-8
CREATE/FDL command • 9-95
/CREATE qualifier
  LIBRARY command • 4-8
/CROSS_REFERENCE qualifier
  FORTRAN/LIST • 4-38
CTRL/C • 8-42
CTRL/Y • 8-42
CTRL/Z • 8-8, 8-64
Current time • 6-44
Cursor position
  on screen • 8-26

D

D_floating format • 6-5
Data
  aligning • 1-23, 9-19
  array (passing) • 1-17
  byte (passing) • 1-24
  character • 6-12
  character (passing) • 1-16
  control information • 1-13
  conversion • 6-20
  defining • 2-3, 6-1
  initializing • 2-5, 6-3
  interprocess • 3-42
  logical • 6-10
  longword (passing) • 1-25
  mask (passing) • 1-26, 1-30
  numeric • 6-4
Data (cont'd.)
  passing • 1-8
  quadword (passing) • 1-28
  record structure • 6-31
  referencing • 6-1
  relation • 6-11
  sharing • 3-42
  storage • 2-3
  structure • 6-1
  untyped • 6-16
  variable-length (passing) • 1-16
  variable-length structure (passing) • 1-31
  word (passing) • 1-24
Database
  compressing • 9-60
  expanding • 9-67
  I/O • 9-16
  record • 9-26
Data compression facility • 9-59
DATA statement • 2-5, 6-3
  array processing • 6-26
Data type
  complex number • 6-6
  debugger • 5-33
  declaration • 2-3
  declaration statement • 6-1
  integer • 6-4
  logical • 6-10
  real number • 6-5
Data type declaration • 2-3
DCL commands
  ANALYZE/RMS_FILE command • 9-94
  CONVERT/FDL • 9-96
  CREATE/FDL command • 9-95
  EDIT/FDL command • 9-93
  LIBRARY command • 9-71
  SET COMMAND command • 7-11
DCL command table • 7-12
DCLTABLES.EXE • 7-12
DCX routines • 9-59
Debugger
  abbreviating commands • 5-4
  breakpoints • 5-18
  command procedures • 5-35
  controlling execution • 5-14, 5-18
  control structures • 5-22

Debugger (cont'd.)
  CTRL/Y • 5-3
  data types • 5-33
  defining keys • 5-5
  displaying source • 5-21
  displays • 5-10
  examining data • 5-30
  global symbols • 5-24
  initialization files • 5-37
  interrupting • 5-3
  invoking • 5-2
  local symbols • 5-23
  log files • 5-38
  path-name • 5-27
  screen mode • 5-7
  storage allocation • 5-25
  suspending execution • 5-18
  terminating • 5-4
  tracepoints • 5-18
  tracing execution • 5-18
  watchpoints • 5-20
  window displays • 5-7
  windows • 5-10
Debugging
  condition handlers • 10-27
  exit handler • 10-39
/DEBUG qualifier
  FORTRAN command • 4-5
  LINK command • 4-9
Default
  library
    system • 4-22
    user • 4-22
  parameter value • 7-3
  qualifier • 7-5
Define
  character string • 6-12
  data • 6-1
  subcommand • 7-22
DEFINE/KEY command • 5-5
DEFINE command • 5-4, 5-28, 7-1
DEFINE VERB statement • 7-1
Defining keys
  in debugger • 5-5
Definition library • 4-24
Definition path • 7-10
Definition statement • 2-3
Delete
  characters from screen • 8-30
  command • 7-14
  pasteboard • 8-17
  virtual display • 8-23
DELETE/KEY command • 5-7
Delete module
  object library • 4-8
Delta time • 6-43
DEPOSIT command • 5-33
Detached process
  creating • 3-7
Device type • 8-61
DIMENSION statement • 6-23
Direct access • 9-4, 9-43
Direct input/output operation • 6-40
Directive
  .FACILITY • 10-9
DISALLOW clause • 7-9
Dispatch to subprogram • 7-26
Display
  deleting • 5-11
  overlying • 5-12
  saving • 5-11
DISPLAY command • 5-10
Divide by 0 error • 6-10
Division • 6-7
DO loop
  array processing • 6-25, 6-29
  implied
    array processing • 6-25, 6-30
    DATA statement • 6-26
DO statement • 2-19, 2-21
Double height • 8-27
DOUBLE PRECISION data type • 6-5
Double spacing • 8-14
Double width • 8-27, 8-28
DO WHILE statement • 2-21, 6-11
Draw
  lines on screen • 8-29
Dummy argument • 1-8
  assumed-size • 6-12

E

Echo
  terminal • 8-50
  terminator • 8-32
EDIT/FDL command • 9-93
Editor
  File Definition Language • 9-93
Elapsed time • 6-40
END DO statement • 2-21
END IF statement • 2-16
End of file • 8-8
END statement • 2-1
Entry point • 4-37
ENTRY statement
  array processing • 6-26
  record processing • 6-38
.EQ. operator • 6-8
EQUAL keyword
  GSMATCH option • 4-30
EQUIVALENCE statement • 2-7
  array processing • 6-26
Equivalent variable • 2-7
.EQV. logical operator • 6-11
Erase
  characters from screen • 8-30
  virtual display • 8-17
Error
  arithmetic • 6-10
  parsing subcommand • 7-25
  see Condition handling • 10-1
  source program • 4-6
Error handling
  general • 10-1
  See Condition Handling • 10-1
ERRSN condition code • 10-17
ERRSNS intrinsic program • 10-17
Escape sequence
  read • 8-64
EVALUATE command • 5-32
Event flag • 3-18
  common • 3-18
  LIB$FREE_EF • 3-19
  LIB$GET_EF • 3-19
  local • 3-18
Event flag (cont'd.)
  manipulating • 3-18
EXAMINE command • 5-31
Executable image • 4-8
Execute
  image • 4-9
EXE file type • 9-71
Exit
  exit handler • 10-35
  image • 10-35
EXIT command • 5-4
Exit handler • 8-64, 10-35
  debugging • 10-39
  establishing • 10-36
  writing • 10-38
Explicit
  data definition • 6-1
Exponentiation • 6-7
Expression • 6-2
  numeric • 6-7
EXTERNAL statement • 1-8, 4-20
Extract module
  object library • 4-8

F

F_floating format • 6-5, 6-6
FAB structure • 9-97
.FACILITY directive • 10-9
.FALSE. operator • 6-11
False value • 6-10
/FAO_COUNT qualifier
  Message Utility • 10-12
FAO argument
  signaling • 10-15
FAO parameter
  specifying • 10-15
Fault
  page • 6-40
FDL (File Definition Language) • 9-92
  applying source • 9-95
  creating source • 9-93
  editor • 9-93
  generating source • 9-94
FDL$CREATE • 9-95

FDL$GENERATE • 9-94
File
  access • 9-3
  attributes • 9-2
  closing • 9-15
  compressing • 9-60
  creating • 9-9
  deleting • 9-6
  direct access • 9-43
  examining attributes • 9-14
  expanding • 9-67
  I/O • 9-16
  indexed • 9-3, 9-43
  interprocess • 3-42
  ISAM • 9-49
  mapping • 9-19
  merging • 9-37
  modifying • 9-96
  name • 9-3
  network • 9-11
  opening • 9-9
  organization • 9-3
  protection • 9-7
  relative • 9-3
  rewinding • 9-14
  scratch • 9-6
  sequential • 9-3, 9-27
  sequential access • 9-43
  sharing • 3-42
  sorting • 9-32
File terminator • 8-64
Fixed-length character string • 6-12
Fixed-length record • 9-4
Flag • 6-19
  see Event flag
  user-defined • 1-13
FOR$ACCEPT • 8-3
FOR$PRINT • 8-3
FOR$READ • 8-3
FOR$TYPE • 8-3
FOR command • 5-22
FOR file type • 4-2
Format
  numeric data • 8-12
  source program • 4-2
Formatted
  conversion • 6-21
  input/output • 6-22
Form feed • 8-14
FORSYSDEF.TLB • 4-24
FORTRAN
  system definition library • 4-24
FORTRAN command • 4-4
FORTRAN I/O errors • 10-16
FORTRAN statements
  COMMON statement • 1-10, 2-8
  DATA statement • 2-5
  DO statement • 2-19, 2-21
  DO WHILE statement • 2-21
  END DO statement • 2-21
  END IF statement • 2-16
  END statement • 2-1
  EQUIVALENCE statement • 2-7
  EXTERNAL statement • 1-8
  FUNCTION statement • 1-4, 2-1
  IF statement • 2-15
  IMPLICIT NONE statement • 2-4
  IMPLICIT statement • 2-4
  INTRINSIC statement • 1-8
  PARAMETER statement • 2-6
  PROGRAM statement • 2-1
  SUBROUTINE statement • 1-4, 2-1
/FULL qualifier
  LINK/MAP command • 4-40
Function
  condition handler • 10-21
  intrinsic • 6-9
    character • 6-15
  reference • 6-2
  system • 6-9
Function code
  write operation • 8-59
Function reference
  array processing • 6-26
FUNCTION statement • 1-4, 2-1, 4-19
  array processing • 6-26
  record processing • 6-38
Function value • 1-4
  system-defined procedure • 1-7

G

G_floating format • 6-5, 6-6
/G_FLOATING qualifier
  FORTRAN command • 6-5
.GE. operator • 6-8
Generic name • 1-5
GLOBAL qualifier • 7-6
Global section • 3-45
  permanent • 3-50
  temporary • 3-50
Global symbol • 4-19
  control information • 1-13
  debugger • 5-24
  resolving • 4-21
GO command • 5-15
GSMATCH option • 4-26
.GT. operator • 6-8

H

H_floating format • 6-5
Header
  library • 9-88
  library module • 9-86
Height
  double • 8-27
Help
  display text
    LBR$ procedure • 9-89
  library • 9-71
HELP command • 5-4
/HELP qualifier
  LIBRARY command • 9-71
Hexadecimal
  constant • 6-17
/HIDE qualifier
  DISPLAY command • 5-12
HLB file type • 9-71
HLP file type • 9-71
Hollerith
  constant • 6-19

I

I/O
  array • 9-17
  attributes • 9-11
  conversion • 6-22
  database • 9-16
  error handling • 10-16
  list-directed • 6-22
  record • 9-17
  specifiers • 9-11
I/O operation
  status of • 8-59
/I4 qualifier
  FORTRAN command • 6-4, 6-10
If state
  composed input • 8-37
IF statement • 2-15, 6-11
IF THEN ELSE command • 5-22
Image
  See Executable image
  command • 7-1
  exiting • 10-35
  installed • 4-10
  map • 4-40
  privileged • 4-10
  shareable • 4-25
IMAGELIB.OLB • 4-22
Image run-down • 10-35
Imaginary
  complex number • 6-6
Implicit
  data definition • 6-2
  variable • 4-7
IMPLICIT NONE statement • 2-4, 4-7
IMPLICIT statement • 2-4, 6-2
Implied
  Input/output unit • 8-3
Implied DO loop • 8-13
  array processing • 6-25, 6-30
  DATA statement • 6-26
INCLUDE statement • 4-24
Indexed file • 9-3
Indicative value • 1-15

Initialization file
  debugger • 5-37
Initialize
  data variable • 6-3
Input
  command • 7-1
  one line • 8-5
  unsolicited • 8-46
Input/output
  array processing • 6-25, 6-30
  asynchronous • 8-58
  buffered operation • 6-40
  channel • 8-55, 8-59
  direct operation • 6-40
  internal • 6-21
  synchronous • 8-55
  user • 8-1
Input/output statement
  array processing • 6-26
  record processing • 6-38
Input operation
  one record • 8-6
INQUIRE statement • 9-14
Insert
  characters on screen • 8-27, 8-28
Install
  privileged image • 4-10
Integer • 6-4
  constant • 6-4
  quadword • 6-5
INTEGER*2 data type • 6-4
INTEGER*4 data type • 6-4
INTEGER data type • 6-4
Internal
  READ statement • 6-21
  WRITE statement • 6-21
Internal parsing • 7-22
Interprocess communication • 3-25
Intrinsic character function • 6-15
Intrinsic function
  CHAR • 6-15
  generic name • 1-5
  specific name • 1-5
INTRINSIC statement • 1-8, 4-20
Intrinsic subprogram • 1-5, 6-9
Intrinsic subprogram (cont'd.)
  generic name • 1-5
  specific name • 1-5
ISAM file • 9-49
Iterative logic • 2-19
  DO loop • 2-20
  indexed DO • 2-21
  nesting • 2-22

K

Key
  alternate • 9-44
  defining • 8-37
  duplicate • 9-58
  multiple • 9-52
  primary • 9-44
  sorting • 9-31
  specifying • 9-43
Keyed access • 9-4
Keypad
  reading from • 8-34
Keyword • 7-7
  ambiguous • 7-17
KEYWORD clause • 7-10
Keyword path • 7-10, 7-17
Keyword specifier • 9-12

L

Label
  formatting source program • 4-2
  parameter • 7-2
  qualifier • 7-4
Last-chance exception vectors • 10-19
LBR$_KEYNOTFND • 9-76
LBR$CLOSE • 9-72
LBR$DELETE_DATA • 9-79
LBR$DELETE_KEY • 9-79
LBR$GET_HEADER • 9-88
LBR$GET_INDEX • 9-91
LBR$GET_RECORD • 9-80
LBR$INI_CONTROL • 9-72
LBR$INSERT_KEY • 9-76
LBR$LOOKUP_KEY • 9-76, 9-79, 9-80, 9-86

LBR$OPEN • 9-72
LBR$OUTPUT_HELP • 9-89
LBR$PUT_END • 9-76
LBR$PUT_RECORD • 9-76
LBR$REPLACE_KEY • 9-77
LBR$SET_MODULE • 9-86
.LE. operator • 6-8
LEQUAL keyword
  GSMATCH option • 4-30
LIB$ADDX • 6-9, 6-44
LIB$DATE_TIME • 6-44
LIB$DAY • 6-46
LIB$DEC_OVER • 10-34
LIB$FLT_UNDER • 10-34
LIB$FREE_EF • 3-19
LIB$FREE_TIMER • 6-41
LIB$FREE_VM • 2-9
LIB$GET_EF • 3-19
LIB$GET_INPUT • 8-5
LIB$GET_LUN • 8-2, 9-3
LIB$GET_VM • 2-9
LIB$INIT_TIMER • 6-40
LIB$INSERT_KEY • 9-83
LIB$INT_OVER • 10-34
LIB$MATCH_COND • 10-24
LIB$PUT_OUTPUT • 2-13, 8-11
LIB$SET_INDEX • 9-83
LIB$SHOW_TIMER • 6-40
LIB$SHOW_VM • 2-13
LIB$SIG_TO_RET
  establishing • 10-7
LIB$SIGNAL
  invoking • 10-6
LIB$SPAWN • 3-2
LIB$STAT_TIMER • 6-41
LIB$STAT_VM • 2-13
LIB$SUBX • 6-9, 6-44
Library
  adding module with LBR$ procedure • 9-76
  closing
    LBR$ procedure • 9-72
  closing with LBR$ procedure • 9-72
  compressing • 9-59
  creating with LBR$ procedure • 9-72
Library (cont'd.)
  deleting module with LBR$ procedure • 9-79
  expanding • 9-59
  header • 9-88
  initializing with LBR$ procedure • 9-72
  inserting module with LBR$ procedure • 9-76
  macro • 4-23
  message • 10-12
  multiple indexes • 9-83
  multiple keys • 9-83
  object • 4-8, 4-21
  opening with LBR$ procedure • 9-72
  processing index entry with LBR$ procedure • 9-91
  replacing module • 9-77
  shareable image • 4-31
  system default • 4-22
  user default • 4-22
LIBRARY command • 4-8, 9-71
Library module
  extracting with LBR$ procedure • 9-80
  header • 9-86
/LIBRARY qualifier
  LINK command • 4-8
Line
  input • 8-5
  output • 8-10
Line editing
  inhibit • 8-52
Link
  program • 4-8
  shareable image • 4-27
LINK/SHAREABLE command • 3-43
LIS file type • 4-5
List
  parameter • 7-3
Listing • 4-33
  compilation • 4-5
  image map • 4-40
  machine code • 4-35
  source • 4-34
List module
  object library • 4-8

/LIST qualifier
  FORTRAN command • 4-5, 4-33
Local event flag • 3-18
LOCAL qualifier • 7-6
Local storage • 2-3
Local symbol • 4-19
  debugger • 5-23
Location
  virtual memory • 4-18
%LOC built-in function • 1-33
Lock manager
  terminal input/output • 3-5
Log file
  debugger • 5-38
Logic
  conditional • 2-14
  iterative • 2-19
  linear • 2-13
  serial • 2-13
Logical
  constant • 6-11
  data type • 6-10
  operator • 6-11
LOGICAL*1 data type • 6-4, 6-10
LOGICAL*2 data type • 6-10
LOGICAL*4 data type • 6-10
LOGICAL data type • 6-10
Logical expression
  evaluation • 6-12
Logical name • 3-25
Logical operator • 2-14
Logical unit number • 8-2, 9-2
Longword
  aligning • 1-23
  in quadword • 1-28
  mask • 1-26
  mask (reading) • 1-35
  passing • 1-25
Loop structure • 2-20
Lowercase
  conversion to uppercase • 8-52
.LT. operator • 6-8

M

MAC file type • 9-71
/MACHINE_CODE qualifier
  FORTRAN/LIST • 4-35
Machine code • 4-35
Macro
  library • 9-71
Macro library • 4-23
/MACRO qualifier
  LIBRARY command • 9-71
Mailbox
  creating • 3-29
  temporary • 3-29
/MAP qualifier
  LINK command • 4-40
Mask • 6-19
  passing • 1-26, 1-30
  reading • 1-35
  setting • 1-26, 1-30
  user-defined • 1-14
Mechanism array • 10-22
MERGE command
  file interface • 9-37
  record interface • 9-38
Message
  see also Message Utility
  chainging • 10-30
  displaying • 10-29
  library • 10-12
  logging • 10-32
Message text
  specifyling variables in • 10-12
Message Utility • 10-9
  compiling message file • 10-12
  .END • 10-10
  .FACILITY • 10-10
  facility name • 10-10
  facility number • 10-10
  message text • 10-11
  message text variables • 10-12
  .SEVERITY • 10-11
  /FAO_COUNT • 10-12
  source file • 10-9
  .TITLE • 10-12

MLB file type • 9-71
Modularity
  virtual displays • 8-40
Module
  name • 9-71
Multidimensional array • 6-28
Multiple output record • 8-13
Multiplication • 6-7

N

Name
  symbol • 4-19
Named constant • 2-6
.NE. operator • 6-8
Negatable qualifier • 7-5
.NEQV. logical operator • 6-11
Nesting • 2-22
Network
  connection • 3-55
  exchanging messages • 3-55
  file access • 9-11
  terminating connection • 3-58
Nokeyword specifier • 9-11
Nonprintable character • 6-15
/NOOPTIMIZE qualifier
  FORTRAN command • 4-5
.NOT. logical operator • 6-11
Number conversion
  to character • 6-22
Numeric
  array • 6-23
    defining • 6-23
    referencing • 6-24
  conversion • 6-20
  expression • 6-7
Numeric data • 6-4
  formatting • 8-12

O

Object library • 4-8, 4-21, 9-71
  add module • 4-8
  create • 4-8
  delete module • 4-8
Object library (cont'd.)
  extract module • 4-8
  list modules • 4-8
Object module • 4-4
/OBJECT qualifier
  FORTRAN command • 4-5
  LIBRARY command • 9-71
OBJ file type • 4-4, 9-71
Octal
  constant • 6-18
OLB file type • 4-8, 4-31, 9-71
OPEN command
  END specifier • 10-16
  ERR specifier • 10-16
  IOSTAT specifier • 10-16
OPEN statement • 8-4, 9-9
  ACCESS specifier • 9-4
  BLOCKSIZE specifier • 9-9
  BUFFERCOUNT specifier • 9-9
  DEFAULTFILE specifier • 9-9
  DISPOSE specifier • 9-6
  EXTENDSIZE specifier • 9-8
  FILE specifier • 9-3
  FORM specifier • 9-5
  INITIALSIZE specifier • 9-8
  IOSTAT specifier • 9-10
  KEY specifier • 9-43
  ORGANIZATION specifier • 9-3
  READONLY specifier • 9-7
  RECL specifier • 9-43
  SHARED specifier • 9-7
  STATUS specifier • 9-6
  UNIT specifier • 9-2
  USEROPEN specifier • 9-97
Operation
  arithmetic • 6-7
Operator
  logical • 6-11
  relational • 6-8
/OPTIMIZE qualifier
  FORTRAN command • 4-5
Options
  creating with LBR$OPEN • 9-72
Options file • 4-26, 4-31
/OPTIONS qualifier
  LINK command • 4-30

.OR. logical operator • 6-11
Out-of-bounds trap • 4-5
Output
  record size • 8-10
  writing • 8-10, 8-11
Output record
  multiple • 8-13
Overflow error • 6-10
Overflow trap • 4-5
Overprinting • 8-15

P

Page
  aligning • 1-23
  fault • 6-40
Parameter • 7-2
  default value • 7-3
  label • 7-2
  list • 7-3
  name • 7-2
  optional • 7-3
  presence • 7-15
  prompt • 7-3
  prompting • 7-4
  required • 7-3
  value • 7-18
PARAMETER clause • 7-2
PARAMETER statement • 2-6, 4-19, 6-2
  assumed-size • 6-12
Parentheses (( ))
  operations • 6-7
Parsing
  command
  subcommand • 7-23
Passed-length character string • 1-16
Passing mechanisms • 1-22
Pasteboard
  delete • 8-17
Path
  definition • 7-10
  keyword • 7-10
Path-name
  debugger • 5-27
Per-Process common block • 3-50
Pop
  virtual display • 8-40
Position
  cursor on screen • 8-26
Positional qualifier • 7-20
Presence
  parameter • 7-15
  qualifier • 7-15
Primary exception vector • 10-18
Privilege
  for images • 4-10
  SS$_NOPRIV • 10-4
Procedure
  system-defined • 1-6
Process • 3-12
  creating • 3-1
  detached • 3-7
  examine • 3-12
  monitor • 3-16
  scheduling • 3-17
Program • 1-2
  building • 4-11
  comiling • 4-1
  command • 7-1
  compiling • 4-4
  control flow • 1-3
  creating • 4-2
  debugging • 3-5
  execution • 3-9, 3-18
  invoking • 1-2, 7-1
  linking • 4-1, 4-8
Program execution
  specified time • 3-9
  timed intervals • 3-10
Program section • 4-37
  aligning data • 1-23
PROGRAM statement • 4-19
Program unit
  structure of • 2-1
Prompt • 8-5, 8-15
  parameter • 7-3
PSECT
  see Program section

Q

Quadword • 6-9
  aligning • 1-23
  mask • 1-30
  mask (reading) • 1-35
  passing • 1-28
Qualifier • 7-4
  combining • 7-9
  default • 7-5
  defining • 7-4
  global • 7-6
  label • 7-4
  local • 7-6, 7-20
  negatable • 7-5
  positional • 7-6, 7-20
  presence • 7-15
  value • 7-6, 7-18
Quotas
  SS$_EXQUOTA • 10-4

R

RAB structure • 9-97
Read
  composed input • 8-37
  keypad input • 8-34
  menu • 8-31
  one input record • 8-6
  screen • 8-32
  screen contents • 8-31
  subcommand • 7-23
Reading
  command
READ statement • 8-6
  internal • 6-21
  keyed • 9-47
Real
  complex number • 6-6
REAL*16 data type • 6-5
REAL*4 data type • 6-5
REAL*8 data type • 6-5
REAL data type • 6-5
Real number • 6-5
Real number (cont'd.)
  constant • 6-5
Record
  accessing • 9-43
  compressing • 9-60
  creating variable-length structure • 1-31
  data structure • 6-31
  deleting • 9-57
  expanding • 9-67
  fixed-length • 9-4
  I/O • 9-17, 9-26
  key • 9-31, 9-43
  merging • 9-38
  segmented • 9-5
  size • 8-10
  sorting • 9-33
  structure • 9-4
  updating • 9-56
  variable-length • 9-5
Record terminator • 8-64
Reference
  character string • 6-13
  data • 6-1
Relation
  data • 6-11
Relational operator • 6-8
Relative file • 9-3
/REMOVE qualifier
  DISPLAY command • 5-11
/REPLACE qualifier
  LIBRARY command • 4-8
Resignaling • 10-25
Return status • 10-3
  from signal • 10-7
Reverse video • 8-18
REWIND statement • 9-14
RMS structures • 9-97
RUN command • 4-9
Run-Time Library procedure • 1-6
  return status • 10-3

S

SAVE statement
  array processing • 6-26

Scratch file
  creating • 9-6
Screen
  special effects • 8-18
Screen debugging • 5-7
Screen management • 8-15
Scroll
  backward • 8-28
  down • 8-28
  forward • 8-28
  output • 8-28
  up • 8-28
SEARCH command • 5-21
Secondary exception vector • 10-18
Section
  deleting • 9-25
  global • 3-45
  mapping • 9-19
  private • 9-19
  updating • 9-25
Segmented record • 9-5
Sequential access • 9-4, 9-43
Sequential file • 9-3
Serial execution • 2-13
SET BREAK command • 5-18
SET COMMAND command • 7-11
SET DISPLAY command • 5-13
SET LOG command • 5-38
SET MODE command • 5-5, 5-8, 5-34
SET MODULE command • 5-24
SET OUTPUT command • 5-36, 5-38
SET SCOPE command • 5-28
SET TRACE command • 5-18
SET TYPE command • 5-33
SET WATCH command • 5-20
Shareable image • 4-25
  create • 4-25
  library • 4-31, 9-71
  linking • 4-31
/SHAREABLE qualifier
  LIBRARY command • 4-31
  LINK command • 4-25, 4-27
/SHARE qualifier
  LIBRARY command • 9-71
SHOW CALLS command • 5-17
SHOW DISPLAY command • 5-9
SHOW KEY command • 5-7
SHOW MODULE command • 5-24
/SHOW qualfifier • 4-34
SHOW SYMBOL command • 5-29
Signal array • 10-21
Signaling • 10-6
  changing to return status • 10-7
  resignaling • 10-25
Single spacing • 8-14
SMG$ADD_KEY_DEF • 8-37
SMG$CHANGE_VIRTUAL_DISPLAY • 8-24
SMG$CHECK_FOR_OCCLUSION • 8-21
SMG$CREATE_KEY_TABLE • 8-37
SMG$CREATE_PASTEBOARD • 8-16
SMG$CREATE_VIRTUAL_DISPLAY • 8-16
SMG$CREATE_VIRTUAL_KEYBOARD • 8-32
SMG$DELETE_CHARS • 8-30
SMG$DELETE_LINE • 8-30
SMG$DELETE_PASTEBOARD • 8-17
SMG$DELETE_VIRTUAL_DISPLAY • 8-23
SMG$DRAW_LINE • 8-29
SMG$DRAW_RECTANGLE • 8-29
SMG$ERASE_CHARS • 8-30
SMG$ERASE_DISPLAY • 8-30
SMG$ERASE_LINE • 8-30
SMG$ERASE_PASTEBOARD • 8-17
SMG$HOME_CURSOR • 8-26
SMG$INSERT_CHARS • 8-27
SMG$INSERT_LINE • 8-28
SMG$LABEL_BORDER • 8-19
SMG$PASTE_VIRTUAL_DISPLAY • 8-16
SMG$POP_VIRTUAL_DISPLAY • 8-40
SMG$PUT_CHARS_HIGHWIDE • 8-27
SMG$PUT_LINE • 8-28
SMG$PUT_LINE_WIDE • 8-28
SMG$PUT_WITH_SCROLL • 8-28
SMG$READ_COMPOSED_LINE • 8-37
SMG$READ_FROM_DISPLAY • 8-31
SMG$READ_STRING • 8-32
SMG$RESTORE_PHYSICAL_SCREEN • 8-40
SMG$RETURN_CURSOR_POS • 8-26
SMG$SAVE_PHYSICAL_SCREEN • 8-40
SMG$SCROLL_DISPLAY_AREA • 8-28

SMG$SET_CURSOR_ABS • 8-26
SMG$SET_CURSOR_REL • 8-26
SMG$SET_DISPLAY_SCROLL_REGION • 8-28
SMG$SET_PHYSICAL_CURSOR • 8-26
SMG$UNPASTE_VIRTUAL_DISPLAY • 8-23
SOR$BEGIN_MERGE • 9-37
SOR$BEGIN_SORT • 9-32
SOR$END_SORT • 9-32
SOR$PASS_FILES • 9-32, 9-37
SOR$RELEASE_REC • 9-33
SOR$RETURN_REC • 9-33
SOR$SORT_MERGE • 9-32
SORT command
  file interface • 9-32
  record interface • 9-33
Source program
  creating • 4-2
  error • 4-6
  formatting spaces • 4-2
  listing • 4-34
SPAWN command • 5-3
Specific name • 1-5
Specifier
  I/O attributes • 9-12
  keyword form • 9-12
  nokeyword form • 9-12
SS$_SUBRNG error • 6-27
STARLET.OLB • 4-22
Start-up file
  debugger • 5-37
Statement
  formatting source program • 4-2
Statistics
  timer • 6-40
Status block
  write operation • 8-60
STEP command • 5-15
Storage • 2-9
  array • 6-27
  character data • 2-11
  contiguous • 2-8
  dynamic • 2-9, 2-10
  large data structures • 2-9
  local • 2-3
Storage (cont'd.)
  numeric data • 2-10
  statistics • 2-13
Storage map • 4-36
String
  See Character string
  counted • 6-15
Subcommand • 7-22
  define • 7-22
  read • 7-23
Subprocess
  concurrent execution • 3-4
  context • 3-3
  creating • 3-2
  delete • 3-7
  execution • 3-4
  in line execution • 3-4
  LIB$SPAWN • 3-2
  SYS$CREPRC • 3-2
Subprogram • 1-1
  as argument • 1-8
  BLOCK DATA statement • 1-12
  dispatch • 7-26
  execution • 1-3
  function • 1-4
  intrinsic • 1-5
  Run-Time Library procedure • 1-6
  subroutine • 1-3
  system-defined procedures • 1-6
  system service procedure • 1-6
Subroutine
  invoking • 1-3
  system • 6-9
SUBROUTINE statement • 1-4, 2-1, 4-19
  array processing • 6-26
  record processing • 6-38
Subscript
  array • 6-24
Substring • 6-13
Subtraction • 6-7
Symbol • 6-1
  and logical names • 3-25
  debugger • 5-28
  defining • 4-19
  global • 4-19

Symbol (cont'd.)
  local • 4-19
  name • 4-19
  storage • 4-18
  universal • 4-30
  unresolved • 4-21
Synchronous input/output • 8-55
Syntax
  error • 4-6
Syntax analysis
SYS$ASCTIM • 6-44
SYS$ASSIGN • 8-55
SYS$BINTIM • 6-44
SYS$CREATE • 9-23
SYS$CREMBX • 3-29
SYS$CREPRC • 3-2, 3-7, 3-9
SYS$CRMPSC • 9-19
SYS$DASSGN • 9-25
SYS$DCLEXH • 10-36
SYS$DELPRC • 3-7
SYS$DELTVA • 9-25
SYS$ENQ • 3-5
SYS$ENQW • 3-5
SYS$ERROR • 10-32
SYS$FAO • 6-44
SYS$GETDVI • 8-61
SYS$GETSYI • 6-42
SYS$GETTIM • 6-42, 6-44
SYS$INPUT • 10-32
SYS$MGBLSC • 3-45
SYS$OPEN • 9-23
SYS$OUTPUT • 8-11
SYS$OUTPUT_HELP • 9-72
SYS$PUTMSG • 10-22, 10-29
SYS$QIO • 3-31, 8-55
SYS$QIOW • 3-31, 8-55
SYS$SCHDWK • 3-11
SYS$SETEXV • 10-18
SYS$SHARE • 4-32
SYS$UNWIND • 10-25
SYS$UPDSEC • 9-25
SYS$WAITFR • 3-7
System
  building • 4-11
  procedures • 6-9
System-defined procedure
  argument (byte) • 1-24
  argument (longword) • 1-25
  argument (mask) • 1-26, 1-30
  argument (quadword) • 1-28
  argument (word) • 1-24
  arguments • 1-20
  function value • 1-7
  invoking • 1-7
System service procedure • 1-6
  return status • 10-3
System time • 6-43

T

Table
  command • 7-11
  key definition • 8-37
Terminal
  characteristics • 8-62
  echo • 8-50
  input/output • 3-5
  timeout • 8-51
  unsolicited input • 8-46
Terminate
  composed input • 8-37
Terminator • 8-5, 8-64
  echo • 8-32
  file • 8-64
Text
  library • 9-71
/TEXT qualifier
  LIBRARY command • 9-71
Time • 6-43
  absolute • 6-43
  current • 6-44
  delta • 6-43
  internal format • 6-43
Time manipulation • 6-44
Timeout
  terminal • 8-51
Timer
  statistics • 6-40
TLB file type • 9-71
Traceback handler • 10-6

/TRACEBACK qualifier • 4-9
Transfer vector • 4-25, 4-28
TRM$M_TM_ESCAPE • 8-34
TRM$M_TM_NOECHO • 8-34
TRM$M_TM_TRMNOECHO • 8-32
.TRUE. operator • 6-11
True value • 6-10
TXT file type • 9-71
Type-ahead buffer • 8-48
TYPE command • 5-21
TYPE statement • 8-13

U

UFO
  see User-file open
Underflow error • 6-10
Underflow trap • 4-5
Underline • 8-18
Union • 6-35
UNIVERSAL option
  LINK command • 4-30
Universal symbol • 4-30
Unpaste
  virtual display • 8-23
Unsolicited input • 8-46
Untyped data • 6-16
Unwind condition handler • 10-25
Uppercase
  conversion from lowercase • 8-52
User
  input/output • 8-1
User command table • 7-12
User-defined condition code
  signaling • 10-13
User-file open • 9-23
User-open routine • 9-97

V

Value
  assignment to variable • 6-3
  parameter • 7-18
  qualifier • 7-6, 7-18
VALUE clause • 7-3
Variable • 4-19, 6-1
  associated • 2-7
  default values • 2-5
  defining • 2-3
  equivalent • 2-7
  I/O • 9-17
  implicit • 4-7
  initializing • 2-5, 2-6
  name • 4-7
  reinitializing • 2-6
Variable-length record • 9-5
Variable-length structure
  passing • 1-31
Verb
  see Command
Video attribute • 8-18
Virtual display
  changing • 8-24
  checking occlusion of • 8-21
  delete • 8-23
  erase • 8-17
  removing • 8-23
  unpaste • 8-23
VOLATILE statement
  record processing • 6-38

W

WHILE command • 5-22
Whole number • 6-4
Width
  double • 8-27, 8-28
  screen • 8-17
Word
  in longword • 1-25
  in quadword • 1-28
  passing • 1-24
Write
  output • 8-10
WRITE statement • 8-11
  internal • 6-21
  keyed • 9-45
Writing output • 8-11

X

.XOR. logical operator • 6-11

Guide to Programming
on VAX/VMS
AI–Y503B–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State ________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

Guide to Programming
on VAX/VMS
AI–Y503B–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____
____
____
____
____
____

Did you find errors in this manual? If so, specify the error and the page number.

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State ________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line