OpenVMS Programming Concepts Manual

December 1995

This manual describes the features that the OpenVMS operating system provides to programmers.

Revision/Update Information:	This manual supersedes the *OpenVMS Programming Concepts Manual*, Version 6.0.
Software Version:	OpenVMS Alpha Version 7.0 OpenVMS VAX Version 7.0

Digital Equipment Corporation
Maynard, Massachusetts

December 1995

5841

This document is available on CD–ROM.

Contents

22 Alignment on OpenVMS VAX and Alpha Systems

23 System Security Services

Figures

Tables

Preface

Intended Audience

This manual is intended for system and application programmers. It presumes that its readers have some familiarity with the OpenVMS programming environment, derived from the *OpenVMS Programming Environment Manual* and OpenVMS high-level language documentation.

Document Structure

This manual's chapters provide information about the programming features of the OpenVMS operating system. A list of the chapters and a summary of their content is as follows:

Chapter 1, Process Creation, defines the two types of processes, what constitutes the context of a process, and the modes of execution of a process. It also describes kernel threads and the kernel threads process structure.

Chapter 2, Process Communication, describes communication within a process and between processes.

Chapter 3, Process Control, describes how to use the creation and control of a process or kernel thread for programming tasks. It also describes how to gather information about a process or kernel thread and how to synchronize a program by using time.

Chapter 4, Using Asynchronous System Traps, describes how to use asynchronous system traps (ASTs). It describes access modes and service routines for ASTs and how ASTs are declared and delivered. It also describes the affects of kernel threads on AST delivery.

Chapter 5, System Time Operations, describes the system time format, and the manipulation of date/time and time conversion. It further describes how to obtain and set the current date and time, how to set and cancel timer requests, and how to schedule and cancel wakeups. The Coordinated Universal Time (UTC) system is also described.

Chapter 6, Using Run-Time Library Routines to Access Operating System Components, describes using run-time libraries (RTLs) with system services, the command language interpreter, and allowing high-level programs to use VAX machine instructions. Also, this chapter describes using RTLs to allocate processwide resources to a single process, perform performance evaluation, and control output formatting.

Chapter 7, Run-Time Library Input/Output Operations, describes using RTLs for input/output operations within a program, using SYS$INPUT, and SYS$OUTPUT, as well as LIB$GET_INPUT and LIB$PUT_OUTPUT. Additionally, this chapter describes using the SMG$ routine for managing terminal screens, and for managing screen management routines.

Chapter 8, File Operations, describes file attributes, strategies to access files, and file protection techniques.

Chapter 9, System Service Input/Output Operations, describes using the SYS$QIO and SYS$QIOW system services for establishing quotas, privileges, and protection. It also describes assigning and deassigning I/O channels, queuing requests, and synchronizing I/O completions. This chapter describes how to use logical names and physical device names for I/O operations; how to use device name defaults; how to obtain information about physical devices; and how to allocate devices. Functions such as mounting, dismounting, and initializing disk and tape volumes, along with using mailboxes are explained.

Chapter 10, Logical Name Services, describes how to create and use logical name services, how to use logical and equivalence names, and how to add and delete entries to a logical name table.

Chapter 11, Distributed Name Service (VAX Only), describes the use of the SYS$DNS system service to provide applications with a method to assign networkwide names to system resources such as printers, files, application databases, disks, nodes, and servers.

Chapter 12, Using the Distributed Transaction Manager, describes the use of the DECdtm system services to ensure the integrity and consistency of distributed transactions by implementing a two-phase commit protocol.

Chapter 13, Condition-Handling Routines and Services, describes the OpenVMS Condition Handling facility. It describes VAX system and Alpha system exceptions, arithmetic exceptions, and Alpha system unaligned access traps. It describes the condition value field, exception dispatcher, signaling, and the argument list passed to a condition handler. Additionally, types of condition handlers and various types of action performed by them are presented. This chapter also describes how to write and debug a condition handler, and how to use an exit handler.

Chapter 14, Synchronizing Data Access and Program Operations, describes synchronization concepts and differences between the VAX system and Alpha system synchronization techniques. It presents methods of synchronization such as event flags, asynchronous system traps (ASTs), parallel processing RTLs, and process priorities, and the affects of kernel threads upon synchronization. It also describes using synchronous and asynchronous system services, and how to write applications in a multiprocessing environment.

Chapter 15, Synchronizing Access to Resources, describes the use of the lock manager system services to synchronize access to shared resources. This chapter presents the concept of resources and locks; it also describes the use of the SYS$ENQ and SYS$DEQ system services to queue and dequeue locks.

Chapter 16, Image Initialization, describes how to use the LIB$INITIALIZE routine to initialize an image.

Chapter 17, Shareable Resources, describes how to share data and program code among programs. It defines shareable images; it also defines and describes how to use local and global symbols to share images.

Chapter 18, Creating User-Written System Services, describes how to create user-written system services with privileged shareable images for both VAX systems and Alpha systems.

Chapter 19, Memory Management Services and Routines (VAX Only), describes the use of system services and RTLs of VAX systems to manage memory. It describes the page size and layout of virtual address space of VAX systems. This chapter also describes how to add virtual address space, adjust working sets, control process swapping, and create and manage sections on VAX systems.

Chapter 20, Memory Management Services and Routines (Alpha Only), describes the use of system services and RTLs of Alpha systems to manage memory. It describes the page size and layout of virtual address space of Alpha systems. This chapter also describes how to add virtual address space, adjust working sets, control process swapping, and create and manage sections on Alpha systems.

Chapter 21, Using Run-Time Routines for Memory Allocation, describes how to use RTLs to allocate and free pages and blocks of memory, and how to use RTLs to create, manage and debug virtual memory zones.

Chapter 22, Alignment on OpenVMS VAX and Alpha Systems, describes the importance and techniques of instruction and data alignment.

Chapter 23, System Security Services, describes the system services that establish protection by using identifiers, rights databases, and access control entries. This chapter also describes how to modify a rights list as well as check access protection.

OpenVMS Alpha Support for 64-Bit Addresses

Alpha

As of Version 7.0, the OpenVMS Alpha operating system provides support for 64-bit virtual memory addresses, which makes the 64-bit virtual address space defined by the Alpha architecture available to the OpenVMS Alpha operating system and to application programs. In the 64-bit virtual address space, both process-private and system virtual address space extend beyond 2 GB. By using 64-bit addressing features, programmers can create images that map and access data beyond the previous limits of 32-bit virtual addresses.

Many tools and languages supported by OpenVMS Alpha (including the Debugger, run-time library routines, and DEC C) are enhanced to support 64-bit virtual addressing. Input and output operations can be performed directly to and from the 64-bit addressable space by means of RMS services, the $QIO system service, and most of the device drivers supplied with OpenVMS Alpha systems.

Underlying this are new system services, which allow an application to allocate and manage the 64-bit virtual address space that is available for process private use.

Nonprivileged programs may optionally be modified to exploit 64-bit addressing support. OpenVMS Alpha 64-bit virtual addressing does not affect nonprivileged programs that are not explicitly modified to exploit 64-bit support. Binary and source compatibility of existing nonprivileged programs is guaranteed.

For more information about OpenVMS Alpha 64-bit virtual addressing features, see the *OpenVMS Alpha Guide to 64-Bit Addressing*. ♦

Related Documents

For a detailed description of each run-time library and system service routine mentioned in this manual, see the OpenVMS Run-Time Library documentation and the *OpenVMS System Services Reference Manual*.

You can find additional information about calling OpenVMS system services and Run-Time Library routines in *OpenVMS Programming Interfaces: Calling a System Routine* and in your language processor documentation. You may also find the following documents useful:

- *OpenVMS DCL Dictionary*
- *OpenVMS User's Manual*
- *Guide to OpenVMS File Applications*
- *OpenVMS Guide to System Security*
- *DECnet for OpenVMS Networking Manual*
- OpenVMS Record Management Services documentation
- *OpenVMS Utility Routines Manual*
- *OpenVMS I/O User's Reference Manual*

For additional information on OpenVMS products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

`http://www.openvms.digital.com`

Reader's Comments

Digital welcomes your comments on this manual.

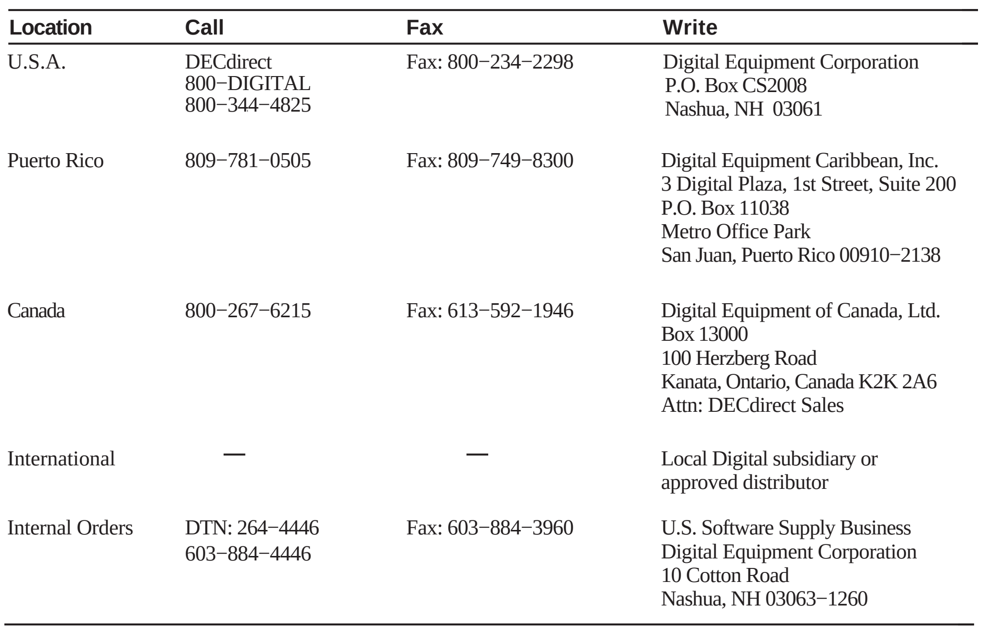
Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OpenVMS Documentation, ZK03-4/U08
Mail	OpenVMS Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders



Location	Call	Fax	Write
U.S.A.	DECdirect 800−DIGITAL 800−344−4825	Fax: 800−234−2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809−781−0505	Fax: 809−749−8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910−2138
Canada	800−267−6215	Fax: 613−592−1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264−4446 603−884−4446	Fax: 603−884−3960	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063−1260

ZK−7654A−GE

Conventions

The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP are synonymous with OpenVMS Alpha or Alpha.

The following conventions are used to identify information specific to OpenVMS Alpha or to OpenVMS VAX:

Alpha	The Alpha icon denotes the beginning of information specific to OpenVMS Alpha.
VAX	The VAX icon denotes the beginning of information specific to OpenVMS VAX.
♦	The diamond symbol denotes the end of a section of information specific to OpenVMS Alpha or to OpenVMS VAX.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x* or GOLD *x*	A sequence such as PF1 *x* or GOLD *x* indicates that you must first press and release the key labeled PF1 or GOLD and then press and release another key or a pointing device button. GOLD key sequences can also have a slash (/), dash (–), or underscore (_) as a delimiter in EVE commands.

Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 Process Creation

This chapter describes process creation and the different types of processes. It also describes kernel threads and the kernel threads process structure. This chapter contains the following sections:

Section 1.1 defines a process and describes the two types of processes.

Section 1.2 describes the execution context of a process.

Section 1.3 describes the modes of execution of a process.

Section 1.4 describes the creation of a subprocess.

Section 1.5 describes the creation of a detached process.

Section 1.6 describes kernel threads and the parts that make up the kernel threads process.

1.1 Process Types

A process is the environment in which an image executes. Two types of processes can be created with the operating system: spawned subprocesses or detached processes.

A **spawned subprocess** is dependent on the process that created it (its parent), and receives a portion of its parent process's resource quotas. The system deletes the spawned subprocess when the parent process exits.

A **detached process** is independent of the process that created it. The process the system creates when you log in is, for example, a detached process. If you want a created process to continue after the parent exits, or not to share resources with the parent, use a detached process.

Table 1–1 compares the characteristics of subprocesses and detached processes.

Table 1–1 Characteristics of Subprocesses and Detached Processes

Characteristic	Subprocess	Detached Process
Privileges	Received from creating process	Specified by creating process
Quotas and limits	Shared with creating process	Specified by creating process, but not shared with creating process
User authorization file	Used for information not given by creating process	Used for most information not given by creating process
User identification code	Received from creating process	Specified by creating process
Restrictions	Exist as long as creating process exists	None

(continued on next page)

Table 1–1 (Cont.) Characteristics of Subprocesses and Detached Processes

Characteristic	Subprocess	Detached Process
How created	SYS$CREPRC, LIB$SPAWN or PPL$SPAWN from another process	SYS$CREPRC from another process
When deleted	At image exit, or when creating process exits	At image exit
Command language interpreter (CLI) present	Usually not if created with SYS$CREPRC; usually yes if spawned	Usually not (though interactive user processes have CLI present, and they are created with SYS$CREPRC)

1.2 Execution Context of a Process

The execution context of a process defines a process to the system. It includes the following:

- Image that the process is executing
- Input and output streams for the image executing in the process
- Disk and directory defaults for the process
- System resource quotas and user privileges available to the process

When the system creates a detached process as the result of a login, it uses the system user authorization file (SYSUAF.DAT) to determine the process's execution context.

For example, the following occurs when you log in to the system:

1. The process created for you executes the image LOGINOUT.
2. The terminal you are using is established as the input, output, and error stream device for images that the process executes.
3. Your disk and directory defaults are taken from the user authorization file.
4. The resource quotas and privileges you have been granted by the system manager are associated with the created process.
5. A command language interpreter (CLI) is mapped into the created process.

1.3 Modes of Execution of a Process

A process executes in one of the following modes:

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

1.4 Creating a Subprocess

You can create a subprocess using the LIB$SPAWN and PPL$SPAWN run-time library routines or the SYS$CREPRC system service. A subprocess created with LIB$SPAWN or PPL$SPAWN is called a spawned subprocess.

Table 1–2 lists the context values provided by LIB$SPAWN, PPL$SPAWN, and SYS$CREPRC for a subprocess when using the default values in the routine calls.

Table 1–2 Comparison of LIB$SPAWN, PPL$SPAWN, and SYS$CREPRC Context Values

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

[1]The created subprocess can include DCL by executing the system image SYS$SYSTEM:LOGINOUT.EXE.

[2]Plus group and job logical name tables.

1.4.1 Using LIB$SPAWN to Create a Spawned Subprocess

The LIB$SPAWN routine enables you to create a subprocess and to set some context options for the new subprocess. LIB$SPAWN creates a subprocess with the same priority as the parent process (generally priority 4). The format for LIB$SPAWN is:

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

For complete information on using each argument, refer to the LIB$SPAWN routine in *OpenVMS RTL Library (LIB$) Manual*.

Specifying a Command String

Use the **command_string** argument to specify a single DCL command to execute once the subprocess is initiated. You can also use this argument to execute a command procedure that, in turn, executes several DCL commands *(@command_procedure_name)*.

Redefining SYS$INPUT and SYS$OUTPUT

Use the **input_file** and **output_file** arguments to specify alternate input and output devices for SYS$INPUT and SYS$OUTPUT. Using alternate values for SYS$INPUT and SYS$OUTPUT can be particularly useful when you are synchronizing processes that are executing concurrently.

Passing Parent Process Context Information to the Subprocess

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

After the Subprocess Completes Execution

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

The LIB$SPAWN routine and SPAWN command do not return a completion status code of 0 from a subprocess command procedure.

Specifying an Alternate Prompt String

Use the **prompt** argument to specify a prompt string for the subprocess.

Specifying an Alternate Command Language Interpreter

Use the **cli** argument to specify a command language interpreter for the subprocess.

Examples of Creating Subprocesses

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

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

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

The following Fortran 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 COMMANDS.COM command procedure. (The CLI$*symbols* are defined in the $CLIDEF module of the system object or in shareable image library. See Chapter 17 for more information.)

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

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

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

The following Fortran 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 Chapter 17.)

```
! 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
```

1.4.2 Using PPL$SPAWN to Create a Spawned Subprocess

The PPL$SPAWN routine works similarly to LIB$SPAWN in that it creates one or more subprocesses with the same context as the parent process on the same node (or system) as the parent process. You can specify the name of an image to be executed in the subprocess. However, you should limit use of PPL$SPAWN to creating subprocesses specifically for parallel processing.

Before using PPL$SPAWN, you must set up special PPL$ data structures with the PPL$INITIALIZE routine; otherwise, unpredictable results may occur. Also, after you create a process with PPLS$CREATE_PROCESS and the process has completed its activity, you must explicitly delete it with PPL$STOP.

For more information about using these PPL$ routines, see the *OpenVMS RTL Parallel Processing (PPL$) Manual*.

1.4.3 Using SYS$CREPRC to Create a Subprocess

The Create Process (SYS$CREPRC) system service creates both subprocesses and detached processes. This section discusses creating a subprocess; Section 1.5 describes creating a detached process. When you call the SYS$CREPRC system service to create a process, you define the context by specifying arguments to the service. The number of subprocesses a process can create is controlled by its PQL$_PRCLM subprocess quota, an individual quota description under the **quota** argument. The DETACH privilege controls your ability to create a detached process with a user identification code (UIC) that is different from the UIC of the creating process.

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

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

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

Ordinarily, when you create a subprocess, you need only assign it an image to execute and, optionally, the SYS$INPUT, SYS$OUTPUT, and SYS$ERROR devices. The system provides default values for the process's privileges, resource quotas, execution modes, and priority. In some cases, however, you may want to define these values specifically. The arguments to the SYS$CREPRC system service that control these characteristics follow. For details, see the descriptions of arguments to the SYS$CREPRC system service in the *OpenVMS System Services Reference Manual*.

The default values passed into the subprocess might not be complete enough for your use. The following sections describe how to modify these default values with SYS$CREPRC.

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

Use the **input**, **output**, and **error** arguments to specify alternate input, output, and error devices for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR. Using alternate values for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR can be particularly useful when you are synchronizing processes that are executing concurrently. By providing alternate or equivalent names for the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR, you can place these logical name /equivalence name pairs in the process logical name table for the created process.

The following C program segment is an example of defining input, output, and error devices for a subprocess:

```
#include <stdio.h>
#include <ssdef.h>
#include <descrip.h>

main() {

 unsigned int status;

 $DESCRIPTOR(input,"SUB_MAIL_BOX");   /* Descriptor for input stream */
 $DESCRIPTOR(output,"COMPUTE_OUT");   /* Descriptor for output and error*/
                                      /* streams */
 $DESCRIPTOR(image,"COMPUTE.EXE");    /* Descriptor for image name */

 /* Create the subprocess */
 status = SYS$CREPRC( 0,              /* process id */
                      &image,         /* image */
                      &input,  1    /* input SYS$INPUT device */
                      &output, 2    /* output SYS$OUTPUT device*/
                      &output, 3    /* error SYS$ERROR device*/
                      0,0,0,0,0,0,0);

}
```

1 The **input** argument equates the equivalence name SUB_MAIL_BOX to the logical name SYS$INPUT. This logical name may represent a mailbox that the calling process previously created with the Create Mailbox and Assign Channel (SYS$CREMBX) system service. Any input the subprocess reads from the logical device SYS$INPUT are read from the mailbox.

2 The **output** argument equates the equivalence name COMPUTE_OUT to the logical name SYS$OUTPUT. All messages the program writes to the logical device SYS$OUTPUT are written to this file.

3 The **error** argument equates the equivalence name COMPUTE_OUT to the logical name SYS$ERROR. All system-generated error messages are written into this file. Because this is the same file as that used for program output, the file effectively contains a complete record of all output produced during the execution of the program image.

The SYS$CREPRC system service does not provide default equivalence names for the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR. If none are specified, any entries in the group or system logical name tables, if any, may provide equivalences. If, while the subprocess executes, it reads or writes to one of these logical devices and no equivalence name exists, an error condition results.

In a program that creates a subprocess, you can cause the subprocess to share the input, output, or error device of the creating process. You must first follow these steps:

1. Use the Get Device/Volume Information (SYS$GETDVI) system service to obtain the device name for the logical name SYS$INPUT, SYS$OUTPUT, or SYS$ERROR.
2. Specify the address of the descriptor returned by the SYS$GETDVI service when you specify the **input**, **output**, or **error** argument to the SYS$CREPRC system service.

This procedure is illustrated in the following example:

```
#include <stdio.h>
#include <ssdef.h>
#include <prcdef.h>
#include <dvidef.h>
#include <descrip.h>

/* Item list to return device name */

struct {
        unsigned short buflen, item_code;
        void *bufaddr;
        void *retlenaddr;
        unsigned int terminator;
}itm_lst;

main() {

        char term[64];
        unsigned int baspri=4, status, *termlen;

/* Descriptors for SYS$GETDVI */
        $DESCRIPTOR(lognam,"SYS$INPUT");

/* Descriptors for SYS$CREPRC */
        $DESCRIPTOR(image,"SYS$SYSTEM:LOGINOUT.EXE");
        $DESCRIPTOR(termdesc, term);

/* Assign values to the item list */

itm_lst.buflen = 64;
itm_lst.item_code = DVI$_DEVNAM;
itm_lst.bufaddr = term;
itm_lst.retlenaddr = &termlen;
itm_lst.terminator = 0;

/* Determine the terminal name */

status = SYS$GETDVI(0,                  /* efn (event flag) */
                    0,                  /* channel */
                    &lognam,            /* devnam */
                    &itm_lst,           /* item list */
                    0,0,0,0);

if((status & 1) != 1)
                    LIB$SIGNAL( status );

/* Create the subprocess */
status = SYS$CREPRC( 0, &image,         /* image to be run */
                     &termdesc,         /* input (SYS$INPUT device) */
                     &termdesc,         /* output (SYS$OUTPUT device) */
                     &termdesc,         /* error (SYS$ERROR device) */
                     0,0,0,
                     &baspri,           /* base priority */
                     0,0,0);
     if((status & 1) != 1)
                     LIB$SIGNAL( status );

}
```

In this example, the subprocess executes, and the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR are equated to the device name of the creating process's logical input device. The subprocess can then do one of the following:

- Use OpenVMS RMS to open the device for reading or writing, or both.
- Use the Assign I/O Channel (SYS$ASSIGN) system service to assign an I/O channel to the device for input/output operations.

In the following example, the program assigns a channel to the device specified by the logical name SYS$OUTPUT:

```
        unsigned int status;
        unsigned short chan;
        $DESCRIPTOR(devnam,"SYS$OUTPUT");
   .
   .
   .
        status = SYS$ASSIGN( &devnam,           /* Device name */
                             &chan,             /* Channel */
                             0, 0, 0);
```

For more information about channel assignment for I/O operations, see Chapter 9.

Setting Privileges

Set different privileges by defining the privilege list for the subprocess using the **prvadr** argument. This is particularly useful when you want to dedicate a subprocess to execute privileged or sensitive code. If you do not specify this argument, the privileges of the calling process are used. If you specify the **prvadr** argument, only the privileges specified in the bit mask are used; the privileges of the calling process are not used. For example, a creating process has the user privileges GROUP and TMPMBX. It creates a process, specifying the user privilege TMPMBX. The created process receives only the user privilege TMPMBX; it does not have the user privilege GROUP.

If you need to create a process that has a privilege that is not one of your current process's privileges, you must have the user privilege SETPRV.

Symbols associated with privileges are defined by the $PRVDEF macro. Each symbol begins with PRV$V_ and identifies the bit number that must be set to specify a given privilege. The following example shows the data definition for a bit mask specifying the GRPNAM and GROUP privileges:

```
struct {
    unsigned int privs = PRV$M_GRPNAM || PRV$M_GROUP;
    unsigned int terminator;
}prvmsk;
```

Setting Process Quotas

Set different process quotas by defining the quota list of system resources for the subprocess using the **quota** argument. This option can be useful when managing a subprocess to limit use of system resources (such as AST usage, I/O, CPU time, lock requests, and working set size and expansion). If you do not specify this argument, the system defines default quotas for the subprocess.

Setting the Subprocess Priority

Set the subprocess priority by setting the base execution priority with the **baspri** argument. If you do not set the subprocess priority, the priority defaults to 2 for VAX MACRO and VAX BLISS–32 and to 0 for all other languages. If you want a subprocess to have a higher priority than its creator, you must have the user privilege ALTPRI to raise the priority level.

Specifying Additional Processing Options

Enable and disable parent and subprocess wait mode, control process swapping, control process accounting, control process dump information, control authorization checks, and control working set adjustments using the **stsflg** argument. This argument defines the status flag, a set of bits that control some execution characteristics of the created process, including resource wait mode and process swap mode.

Defining an Image for a Subprocess to Execute

When you call the SYS$CREPRC system service, use the **image** argument to provide the process with the name of an image to execute. For example, the following lines of C create a subprocess to execute the image named CARRIE.EXE:

```
        $DESCRIPTOR(image,"CARRIE");
   .
   .
   .
        status = SYS$CREPRC(0, &image,  . . .  );
```

In this example, only a file name is specified; the service uses current disk and directory defaults, performs logical name translation, uses the default file type .EXE, and locates the most recent version of the image file. When the subprocess completes execution of the image, the subprocess is deleted. Process deletion is described in Chapter 3.

1.4.3.1 Disk and Directory Defaults for Created Processes

When you use the SYS$CREPRC system service to create a process to execute an image, the system locates the image file in the default device and directory of the created process. Any created process inherits the current default device and directory of its creator.

If a created process runs an image that is not in its default directory, you must identify the directory and, if necessary, the device in the file specification of the image to be run.

There is no way to define a default device or directory for the created process that is different from that of the creating process in a call to SYS$CREPRC. The created process can, however, define an equivalence for the logical device SYS$DISK by calling the Create Logical Name ($CRELNM) system service.

If the process is a subprocess, you, in the creating process, can define an equivalence name in the group logical name table, job logical name table, or any logical name table shared by the creating process and the subprocess. The created process then uses this logical name translation as its default directory. The created process can also set its own default directory by calling the OpenVMS RMS default directory system service, SYS$SETDDIR.

A process can create a process with a default directory that is different from its own by completing the following steps in the creating process:

1. Make a call to SYS$SETDDIR to change its own default directory.
2. Make a call to SYS$CREPRC to create the new process.
3. Make a call to SYS$SETDDIR to change its own default directory back to the default directory it had before the first call to SYS$SETDDIR.

The creating process now has its original default directory. The new process has the different default directory that the creating process had when it created the new process. For details on how to call SYS$SETDDIR, see the *OpenVMS System Services Reference Manual*.

1.4.4 Debugging Within a Subprocess

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

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

1.5 Creating a Detached Process

The creation of a detached process is primarily a task the operating system performs when you log in. In general, an application creates a detached process only when a program must continue executing after the parent process exits. To do this, you should use the SYS$CREPRC system service. You can also use detached processes to write to another process's terminal by using the SYS$BREAKTHRU system service.

The DETACH privilege controls the ability to create a detached process with a UIC that is different from the UIC of the creating process. You can use the **uic** argument to the SYS$CREPRC system service to define whether a process is a subprocess or a detached process. The **uic** argument provides the created process with a user identification code (UIC). If you omit the **uic** argument, the SYS$CREPRC system service creates a subprocess that executes under the UIC of the creating process.

You can also create a detached process with the same UIC as the creating process by specifying the detach flag in the **stsflg** argument. You do not need the DETACH privilege to create a detached process with the same UIC as the creating process.

Examples of Creating a Detached Process

The following Fortran 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 command 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
```

1.6 Kernel Threads and the Kernel Threads Process Structure (Alpha Only)

Alpha

This section defines and describes some advantages of using kernel threads. It also describes some kernel threads features and type of model, as well as the design changes made to the OpenVMS operating system.

Note

For information about the concepts and implementation of user threads with DECthreads, see the *Guide to DECthreads*.

1.6.1 Definition and Advantages of Kernel Threads

A **thread** is a single, sequential flow of execution within a process's address space. A single process contains an address space wherein a single thread or multiple threads execute concurrently. Programs typically have a single flow of execution and therefore a single thread, whereas multithreaded programs have multiple points of execution at any one time.
By using threads as a programming model, you can gain the following advantages:

- More modular code design
- Simpler application design and maintenance
- The potential to run independent flows of execution in parallel on multiple CPUs
- The potential to make better use of available CPU resources through parallel execution

1.6.2 Kernel Threads Features

With kernel threads, the OpenVMS operating system implements the following two features:

- Multiple execution contexts within a process
- Efficient use of the OpenVMS and DECthreads schedulers

1.6.2.1 Multiple Execution Contexts within a Process

Before the implementation of kernel threads, the scheduling model for the OpenVMS operating system was per process. The only scheduling context was the process itself; that is, only one execution context per process. Since a threaded application could create thousands of threads, many of these threads could potentially be executing at the same time. But because OpenVMS processes had only a single execution context, in effect, only one of those application threads was running at any one time. If this multithreaded application was running on a multiprocessor system, the application could not make use of more than a single CPU.

After the implementation of kernel threads, the scheduling model allows for multiple execution contexts within a process; that is, more than one application thread can be executing concurrently. These execution contexts are called kernel threads. Kernel threads allows a multithreaded application to have a thread executing on every CPU in a multiprocessor system. Kernel threads, therefore, allow a threaded application to take advantage of multiple CPUs in a symmetric multiple processing (SMP) system.

1.6.2.2 Efficient Use of the OpenVMS and DECthreads Schedulers

It is the function of the user mode thread manager to schedule individual user mode application threads. On OpenVMS, DECthreads is the user mode threading package of choice. Before the implementation of kernel threads, DECthreads multiplexed user mode threads on the single OpenVMS execution context—the process. DECthreads implemented parts of its scheduling by using a periodic timer. When the AST executed and the thread manager gained control, the thread manager could then select a new application thread for execution. But because the thread manager could not detect that a thread had entered an OpenVMS wait state, the entire application blocked until that periodic AST was delivered. That resulted in a delay until the thread manager regained control and could schedule another thread. Once the thread manager gained control, it could schedule a previously preempted thread unaware that the thread was in a wait state. The lack of integration between the OpenVMS and DECthreads schedulers could result in wasted CPU resources.

After the implementation of kernel threads, the scheduling model provides for scheduler callbacks. A scheduler callback is an upcall from the OpenVMS scheduler to the thread manager whenever a thread changes state. This upcall allows the OpenVMS scheduler to inform the thread manager that the current thread is stalled and that another thread should be scheduled. Upcalls also inform the thread manager that an event a thread is waiting on has completed. With kernel threads, the two schedulers are better integrated, minimizing application thread scheduling delays.

1.6.3 Kernel Threads Model and Design Features

This section presents the type of kernel threads model that OpenVMS Alpha implements, and some features of the operating system design that changed to implement the kernel thread model.

1.6.3.1 Kernel Threads Model

The OpenVMS kernel threads model is one that implements a few kernel threads to many user threads with integrated schedulers. With this model, there is a mapping of many user threads to only several execution contexts or kernel threads. The kernel threads have no knowledge of the individual threads within an application. The thread manager multiplexes those user threads on an execution context, though a single process can have multiple execution contexts. This model also integrates the user mode thread manager scheduler with the OpenVMS scheduler.

1.6.3.2 Kernel Threads Design Features

Design additions and modifications made to various features of OpenVMS Alpha are as follows:

- Process Structure
- Access to Inner Modes
- Scheduling
- ASTs
- Event Flags
- Process Control Services

1.6.3.2.1 Process Structure With the implementation of OpenVMS kernel threads, all processes are a threaded process with at least one kernel thread. Every kernel thread gets a set of stacks, one for each access mode. Quotas and limits are maintained and enforced at the process level. The process virtual address space remains per process and is shared by all threads. The scheduling entity moves from the process to the kernel thread. In general, ASTs are delivered directly to the kernel threads. Event flags and locks remain per process. See Section 1.6.4 for more information.

1.6.3.2.2 Access to Inner Modes With the implementation of kernel threads, a single threaded process continues to function exactly as it has in the past. A multithreaded process may have multiple threads executing in user mode or in user mode ASTs, as is also possible for supervisor mode. Except in cases where an activity in inner mode is considered *thread safe*, a multithreaded process may have only a single thread executing in an inner mode at any one time. Multithreaded processes retain the normal preemption of inner mode by more inner mode ASTs. A special inner mode semaphore serializes access to inner mode.

1.6.3.2.3 Scheduling With the implementation of kernel threads, the OpenVMS scheduler concerns itself with kernel threads, and not processes. At certain points in the OpenVMS executive at which the scheduler could wait a kernel thread, it can instead transfer control to the thread manager. This transfer of control, known as a callback or upcall, allows the thread manager the chance to reschedule stalled application threads.

1.6.3.2.4 ASTs With the implementation of kernel threads, ASTs are not delivered to the process. They are delivered to the kernel thread on which the event was initiated. Inner mode ASTs are generally delivered to the kernel thread already in inner mode. If no thread is in inner mode, the AST is delivered to the kernel thread that initiated the event.

1.6.3.2.5 Event Flags With the implementation of kernel threads, event flags continue to function on a per-process basis, maintaining compatibility with existing application behavior.

1.6.3.2.6 Process Control Services With the implementation of kernel threads, many process control services continue to function at the process level. SYS$SUSPEND and SYS$RESUME system services, for example, continue to change the scheduling state of the entire process, including all of its threads. Other services such as SYS$HIBER and SYS$SCHDWK act on individual kernel threads instead of the entire process.

1.6.4 Kernel Threads Process Structure

This section describes the components that make up a kernel threads process. It describes the following components:

- Process control block (PCB) and process header (PHD)
- Kernel thread block (KTB)
- Floating-point registers and execution data block (FRED)
- Kernel threads region
- Per-kernel thread stacks
- Per-kernel thread data cells
- Process status bits

1.6.4.1 Process Control Block (PCB) and Process Header (PHD)

Two primary data structures exist in the OpenVMS executive that describe the context of a process:

- Software process control block (PCB)
- Process header (PHD)

The PCB contains fields that identify the process to the system. The PCB comprises contexts that pertain to quotas and limits, scheduling state, privileges, AST queues, and identifiers. In general, any information that is required to be resident at all times is in the PCB. Therefore, the PCB is allocated from nonpaged pool.

The PHD contains fields that pertain to a process's virtual address space. The PHD consists of the working set list, and the process section table. The PHD also contains the hardware process control block (HWPCB), and a floating-point register save area. The HWPCB contains the hardware execution context of the process. The PHD is allocated as part of a balance set slot, and it can be outswapped.

1.6.4.1.1 Effect of a Multithreaded Process on the PCB and PHD With multiple execution contexts within the same process, the multiple threads of execution all share the same address space, but have some independent software and hardware context. This change to a multithreaded process results in an impact on the PCB and PHD structures, and on any code that references them.

Before the implementation of kernel threads, the PCB contained much context that was per-process. Now, with the introduction of multiple threads of execution, much context becomes per-thread. To accommodate per-thread context, a new data structure, the kernel thread block (KTB), is created, with the per-thread context removed from the PCB. However, the PCB continues to contain context common to all threads, such as quotas and limits. The new per-kernel thread structure contains the scheduling state, priority, and the AST queues.

The PHD contains the HWPCB which gives a process its single execution context. The HWPCB remains in the PHD; this HWPCB is used by a process when it is first created. This execution context is also called the initial thread. A single threaded process has only this one execution context. A new structure, the floating-point registers and execution data block (FRED), is created to contain the hardware context of the newly created kernel threads. Since all threads in a process share the same address space, the PHD continues to describe the entire virtual memeory layout of the process.

1.6.4.2 Kernel Thread Block (KTB)

The kernel thread block (KTB) is a new per-kernel-thread data structure. The KTB contains all per-thread software context moved from the PCB. The KTB is the basic unit of scheduling, a role previously performed by the PCB, and is the data structure placed in the scheduling state queues. Since the KTB is the logical extension of the PCB, the SCHED spinlock synchronizes access to the KTB and the PCB.

Typically, the number of KTBs a multithreaded process has is the same as the number of CPUs on the system. Actually, the number of KTBs is limited by the value of the system parameter MULTITHREAD. If MULTITHREAD is zero, the OpenVMS kernel support is disabled. With kernel threads disabled, user-level threading is still possible with DECthreads. The environment is identical to the OpenVMS environment prior to the OpenVMS V7.0 release. If MULTITHREAD is non-zero, it represents the maximum number of execution contexts or kernel threads that a process can own, including the initial one.

The KTB, in reality, is not an independent structure from the PCB. Both the PCB and KTB are defined as sparse structures. The fields of the PCB that move to the KTB retain their original PCB offsets in the KTB. In the PCB, these fields are unused. In effect, if the two structures are overlaid, the result is the PCB as it currently exists with new fields appended at the end. The PCB and KTB for the initial thread occupy the same block of nonpaged pool; therefore, the KTB address for the initial thread is the same as for the PCB.

1.6.4.3 Floating-Point Registers and Execution Data Block (FRED)

To allow for multiple execution contexts, not only are additional KTBs required to maintain the software context, but additional HWPCBs must be created to maintain the hardware context. Each HWPCB has allocated with it a block of 256 bytes for preserving the contents of the floating-point registers across context switches. Another 128 bytes is allocated for per-kernel thread data.

The combined structure that contains the HWPCB, floating-point register save area, and per-kernel thread data is called the floating-point registers and execution data (FRED) block. These structures reside in the process's balance set slot. This allows the FREDs to be outswapped with the process header.

1.6.4.4 Kernel Threads Region

Much process context resides in P1 space, taking the form of data cells and the process stacks. Some of these data cells need to be per kernel thread, as do the stacks. During initialization of the multithread environment, a kernel thread region in P1 space is initialized to contain the per-kernel-thread data cells and stacks. The region begins at the boundary between P0 and P1 space at address 40000000x, and it grows toward higher addresses and the initial thread's user stack. The region is divided into per-kernel-thread areas. Each area contains pages for data cells and the four stacks.

1.6.4.5 Per-Kernel Thread Stacks

A process is created with four stacks; each access mode has one stack. All four of these stacks are located in P1 space. Stack sizes are either fixed, determined by a SYSGEN parameter, or expandable. The parameter KSTACKPAGES controls the size of the kernel stack. This parameter continues to control all kernel stack sizes, including those created for new execution contexts of kernel threads. The executive stack is a fixed size of two pages; with kernel threads implementation, the executive stack for new execution contexts continues to be two pages in size. The supervisor stack is a fixed size of four pages; with kernel threads implementation, the supervisor stack for new execution contexts is reduced to two pages in size.

For the user stack, a more complex situation exists. OpenVMS allocates P1 space from high to lower addresses. The user stack is placed after the lowest P1 space address allocated. This allows the user stack to expand on demand toward P0 space. With the introduction of multiple sets of stacks, the locations of these stacks impose a limit on the size of each area in which they can reside. With the implementation of kernel threads, the user stack is no longer boundless. The initial user stack remains semi-boundless; it still grows toward P0 space, but the limit is the per-kernel thread region instead of P0 space.

1.6.4.6 Per-Kernel-Thread Data Cells

Several pages in P1 space contain process state in the form of data cells. A number of these cells must have a per-kernel-thread equivalent. These data cells do not all reside on pages with the same protection. Because of this, the per-kernel-thread area reserves two pages for these cells. Each page has a different page protection; one page protection is user read, user write (URUW); the other is user read, executive write (UREW).

1.6.4.7 Summary of Process Data Structures

Process creation results in a PCB/KTB, a PHD/FRED, and a set of stacks. All processes have a single kernel thread, the initial thread.

A multithreaded process always begins as a single threaded process. A multithreaded process contains a PCB/KTB pair and a PHD/FRED pair for the initial thread; for its other threads, it contains additional KTBs, additional FREDs, and additional sets of stacks. When the multithreaded application exits, the process returns to its single threaded state, and all additional KTBs, FREDs, and stacks are deleted. ♦

2 Process Communication

This chapter describes communication mechanisms used within a process and between processes. It contains the following sections:

Section 2.1 describes communication within a process.

Section 2.2 describes communication between processes.

The operating system allows your process to communicate within itself and with other processes. Processes can be either wholly independent or cooperative. This chapter presents considerations for developing applications that require the concurrent execution of many programs, and how you can use process communication to perform the following functions:

- Synchronize events
- Share data
- Obtain information about events important to the program you are executing

2.1 Communication Within a Process

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

- Local event flags
- Logical names (in supervisor mode)
- Global symbols (command language interpreter symbols)
- Common area

For passing information among chained images, you can use all four methods because the image reading the information executes immediately after the image that deposited it. Only the common area allows you to pass data reliably from one image to another in the event that another image's execution intervenes between the two communicating images.

For communicating within a single image, you can use event flags, logical names, and symbols. For synchronizing events within a single image, use event flags. See Chapter 14 for more information about synchronizing events.

Because permanent mailboxes and permanent global sections are not deleted when the creating image exits, they also can be used to pass information from the current image to a later executing image. However, Digital recommends that you use the common area because it uses fewer system resources than the permanent structures and does not require privilege. (You need the PRMMBX privilege to create a permanent mailbox and the PRMGBL privilege to create a permanent global section.)

2.1.1 Using Local Event Flags

Event flags are status-posting bits maintained by the operating system for general programming use. Programs can set, clear, and read event flags. By setting and clearing event flags at specific points, one program component can signal when an event has occurred. Other program components can then check the event flag to determine when the event has been completed. For more information about using local and common event flags for synchronizing events, refer to Chapter 14.

2.1.2 Using Logical Names

Logical names can store up to 255 bytes of data. When you need to pass information from one program to another within a process, you can assign data to a logical name when you create the logical name; then, other programs can access the contents of the logical name. See Chapter 10 for more information about logical name system services.

2.1.2.1 Using Logical Name Tables

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

2.1.2.2 Using Access Modes

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

2.1.2.3 Creating and Accessing Logical Names

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

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

Example 2–1 creates a spawned subprocess to perform an iterative calculation. The logical name REP_NUMBER specifies the number of times that REPEAT, the program executing in the subprocess, should perform the calculation. Because both the parent process and the subprocess are part of the same job, REP_NUMBER is placed in the job logical name table LNM$JOB. (Note that logical names are case sensitive; specifically, LNM$JOB is a system-defined logical name that refers to the job logical name table, whereas lnm$job is not.) To satisfy the references to LNM$_STRING, the example includes the file $LNMDEF.

Example 2–1 Performing an Iterative Calculation with a Spawned Subprocess

```
PROGRAM CALC

! Status variable and system routines
INTEGER*4 STATUS,
2         SYS$CRELNM,
2         LIB$GET_EF,
2         LIB$SPAWN
! Define itmlst structure
STRUCTURE /ITMLST/
 UNION
  MAP
   INTEGER*2 BUFLEN
   INTEGER*2 CODE
   INTEGER*4 BUFADR
   INTEGER*4 RETLENADR
  END MAP
  MAP
   INTEGER*4 END_LIST
  END MAP
 END UNION
END STRUCTURE
! Declare itmlst
RECORD /ITMLST/ LNMLIST(2)
! Number to pass to REPEAT.FOR
CHARACTER*3 REPETITIONS_STR
INTEGER REPETITIONS
! Symbols for LIB$SPAWN and SYS$CRELNM
! Include FORSYSDEF symbol definitions:
INCLUDE        '($LNMDEF)'
EXTERNAL CLI$M_NOLOGNAM,
2        CLI$M_NOCLISYM,
2        CLI$M_NOKEYPAD,
2        CLI$M_NOWAIT,
2        LNM$_STRING
                 .
                 . ! Set REPETITIONS_STR
                 .
! Set up and create logical name REP_NUMBER in job table
LNMLIST(1).BUFLEN     = 3
LNMLIST(1).CODE       = %LOC (LNM$_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))
```

(continued on next page)

Example 2–1 (Cont.) Performing an Iterative Calculation with a Spawned Subprocess

```
! 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))
```

(continued on next page)

Example 2–1 (Cont.) Performing an Iterative Calculation with a Spawned Subprocess

```
! 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
```

2.1.3 Using Command Language Interpreter Symbols

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

2.1.3.1 Local and Global Symbols

The two kinds of CLI symbols and their definitions are as follows:

- Local—A local symbol is available to the command level that defined it, any command procedure executed from that command level, and lower command levels.
- Global—A global symbol can be accessed from any command level, regardless of the level at which it was defined.

2.1.3.2 Creating and Using Global Symbols

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

To use DCL global symbols, follow this procedure:

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

2.1.4 Using the Common Area

Use the common area to store data from one image to the next. Such data is unlikely to be corrupted between the time one image deposits it in a common area and another image reads it from the area. The common area can store 252 bytes of data. The LIB$PUT_COMMON routine writes information to this common area; the LIB$GET_COMMON routine reads information from this common area.

2.1.4.1 Creating the Process Common Area

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

2.1.4.2 Common I/O Routines

The LIB$PUT_COMMON routine allows a program to copy a string into the process's common storage area. This area remains defined during multiple image activations. LIB$GET_COMMON allows a program to copy a string from the common area into a destination string. The programs reading and writing the data in the common area must agree upon its amount and format. The maximum length of the destination string is defined as follows:

[min(256, the length of the data in the common storage area) - 4]

This maximum length is normally 252.

In BASIC and Fortran, you can use these routines to allow a USEROPEN routine to pass information back to the routine that called it. A USEROPEN routine cannot write arguments. However, it can call LIB$PUT_COMMON to put information into the common area. The calling program can then use LIB$GET_COMMON to retrieve it.

You can also use these routines to pass information between images run successively, such as chained images run by LIB$RUN_PROGRAM.

2.1.4.3 Modifying or Deleting Data in the Common Block

You cannot modify or delete data in the process common area unless LIB$PUT_COMMON is invoked. Therefore, you can execute any number of images between one image and another, provided that LIB$PUT_COMMON has not been invoked. Each subsequent image reads the correct data. Invoking LIB$GET_COMMON to read the common block does not modify the data.

2.1.4.4 Specifying Other Types of Data

Although the descriptions of the LIB$PUT_COMMON and LIB$GET_COMMON routines in the *OpenVMS RTL Library (LIB$) Manual* specify a character string for the argument containing the data written to or read from the common area, you can specify other types of data. However, you must pass both noncharacter and character data by descriptor.

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

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

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

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

! Compile the report
   .
   .
   .
```

2.2 Communication Between Processes

Communication between processes, or interprocess communication, can be performed in the following ways:

- Shared files
- Common event flags
- Logical names
- Mailboxes
- Global sections
- Lock management system services

Each approach offers different possibilities in terms of the speed at which it communicates information and the amount of information it can communicate. For example, shared files offer the possibility of sharing an unlimited amount of information; however, this approach is the slowest because the disk must be accessed to share information.

Like shared files, global sections offer the possibility of sharing large amounts of information. Because sharing information through global sections requires only memory access, it is the fastest communication method.

Logical names and mailboxes can communicate moderate amounts of information. Because each method operates through a relatively complex system service, each is faster than files, but slower than the other communication methods.

The lock management services and common event flag cluster methods can communicate relatively small amounts of information. With the exception of global sections, they are the fastest of the interprocess communication methods.

Common event flags: Processes executing within the same group can use common event flags to signal the occurrence or completion of particular activities. For details about event flags, and an example of how cooperating processes in the same group use a common event flag, see Chapter 14.

Logical name tables: Processes executing in the same job can use the job logical name table to provide member processes with equivalence names for logical names. Processes executing in the same group can use the group logical name table. A process must have the GRPNAM or SYSPRN privilege to place names in the group logical name table. All processes in the system can use the system logical name table. A process must have the SYSNAM or SYSPRV privilege to place names in the system logical name table. Processes can also create and use user-defined logical name tables. For details about logical names and logical name tables, see Chapter 10.

Mailboxes: You can use mailboxes as virtual input/output devices to pass information, messages, or data among processes. For additional information on how to create and use mailboxes, see Section 2.2.1. Mailboxes may also be used to provide a creating process with a way to determine when and under what conditions a created subprocess was deleted. For an example of a termination mailbox, see Section 3.8.4.2.

Global sections: Global sections can be either disk files or page-file sections that contain shareable code or data. Through the use of memory management services, these files can be mapped to the virtual address space of more than one process. In the case of a data file on disk, cooperating processes can synchronize reading and writing the data in physical memory; as data is updated, system paging results in the updated data being written directly back into the disk file. Global page-file sections are useful for temporary storage of common data; they are not mapped to a disk file. Instead, they page only to the system default page file. Global sections are described in more detail in Chapter 19 and Chapter 20.

Lock management system services: Processes can use the lock management system services to control access to resources (any entity on the system that the process can read, write, or execute). In addition to controlling access, the lock management services provide a mechanism for passing information among processes that have access to a resource (lock value blocks). Blocking ASTs can be used to notify a process that other processes are waiting for a resource. Using lock value blocks is a practical technique for communicating in cluster environments. With lock value blocks, communication between two processes from node to node in a distributed environment is an effective way of implementing cluster communication. For more information about the lock management system services, see Chapter 15.

While common event flags and lock management services establish communication, they are most useful for synchronizing events and are discussed in Chapter 14. Global sections and shared files are best used for sharing data and are discussed in Chapter 17.

2.2.1 Mailboxes

A mailbox is a virtual device used for communication among processes. You must call OpenVMS RMS services, language I/O statements, or I/O system services to perform actual data transfers.

2.2.1.1 Creating a Mailbox

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

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

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

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

- Specify a variable to receive the I/O channel number using the **chan** argument. This argument is required.
- Specify the logical name to be associated with the mailbox using the **lognam** argument. The logical name identifies the mailbox for other processes and for input/output statements.

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

2.2.1.2 Creating Temporary and Permanent Mailboxes

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

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

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

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

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

Note

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

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

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

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

END
```

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

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

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

END
```

2.2.1.3 Assigning an I/O Channel Along with a Mailbox

A mailbox is a virtual device used for communication between processes. A channel is the communication path that a process uses to perform I/O operations to a particular device. The LIB$ASN_WTH_MBX routine assigns a channel to a device and associates a mailbox with the device.

Normally, a process calls the SYS$CREMBX system service to create a mailbox and assign a channel and logical name to it. In the case of a temporary mailbox, this service places the logical name corresponding to the mailbox in the job logical name table. This implies that any process running in the same job and using the same logical name uses the same mailbox.

Sometimes it is not desirable to have more than one process use the same mailbox. For example, when a program connects explicitly with another process across a network, the program uses a mailbox to obtain the data confirming the connection and to store the asynchronous messages from the other process. If that mailbox is shared with other processes in the same group, there is no way to determine which messages are intended for which processes; the processes read each other's messages, and the original program does not receive the correct information from the cooperating process across the network link.

The LIB$ASN_WTH_MBX routine avoids this situation by associating the physical mailbox name with the channel assigned to the device. To create a temporary mailbox for itself and other processes cooperating with it, your program calls LIB$ASN_WTH_MBX. The run-time library routine assigns the channel and creates the temporary mailbox by using the system services $GETDVI, $ASSIGN, and $CREMBX. Instead of a logical name, the mailbox is identified by a physical device name of the form *MBcu*. The elements that make up this device name are as follows:

MB	indicates that the device is a mailbox.
c	is the controller.
u	is the unit number.

The routine returns this device name to the calling program, which then must pass the mailbox channel to the other programs with which it cooperates. In this way, the cooperating processes access the mailbox by its physical name instead of by its jobwide logical name.

The calling program passes the routine a device name, which specifies the device to which the channel is to be assigned. For this argument (called **dev-nam**), you can use a logical name. If you do so, the routine attempts one level of logical name translation.

The privilege restrictions and process quotas required for using this routine are those required by the $GETDVI, $CREMBX, and $ASSIGN system services.

2.2.1.4 Reading and Writing Data to a Mailbox

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

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

Chapter 9 describes the SYS$QIO and SYS$QIOW system services and provides further discussion of mailbox I/O. See the *OpenVMS System Services Reference Manual* for more information. Digital Equipment Corporation recommends that you supply the optional I/O status block parameter when you use these two system services. The contents of the status block varies depending on the QIO function code; refer to the function code descriptions in the *OpenVMS I/O User's Reference Manual* for a description of the appropriate status block.

2.2.1.5 Using Synchronous Mailbox I/O

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

The program segment shown in Example 2–2 opens a mailbox for the first time. 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.

Example 2–2 Opening a Mailbox

```
! Status variable and values
INTEGER STATUS

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

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

In Example 2–3, one image passes device names to a second image. The second image returns the process name and the terminal associated with the process that allocated each device. A WRITE statement in the first image does not complete until the cooperating process issues a READ statement. (The variable declarations are not shown in the second program because they are very similar to those in the first program.)

Example 2–3 Synchronous I/O Using a Mailbox

```
! DEVICE.FOR

PROGRAM PROCESS_DEVICE

! Status variable
INTEGER STATUS

! Name and I/O channel for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! Logical unit number for FORTRAN I/O
INTEGER MBX_LUN
```

(continued on next page)

Example 2–3 (Cont.) Synchronous I/O Using a Mailbox

```
! 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
```

(continued on next page)

Example 2–3 (Cont.) Synchronous I/O Using a Mailbox

```
! 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
```

2.2.1.6 Using 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 do not exit until the other process has a channel to the mailbox, or use a permanent mailbox.

Queueing an Immmediate I/O Request

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

Reading Data from the Mailbox

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

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

In Example 2–4, the first program creates a mailbox named MAIL_BOX, writes data to it, and then indicates the end of the data by writing an end-of-file message. The second program creates a mailbox with the same logical name, reads the messages from the mailbox into an array, and stops the read operations when a read operation generates an end-of-file message and the second longword of the I/O status block is nonzero, confirming that the writing process sent the end-of-file message. The processes use common event flag 64 to ensure that SEND.FOR does not exit until RECEIVE.FOR has established a channel to the mailbox. (If RECEIVE.FOR executes first, an error occurs because SYS$ASSIGN cannot find the mailbox.)

Example 2–4 Immediate I/O Using a Mailbox

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

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

(continued on next page)

Example 2–4 (Cont.) Immediate I/O Using a Mailbox

```
! 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
```

(continued on next page)

Example 2–4 (Cont.) Immediate I/O Using a Mailbox

```
! 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
   .
   .
   .
```

2.2.1.7 Using Asynchronous Mailbox I/O

Use asynchronous I/O to queue a read or write request to a mailbox. To ensure that the other process receives the information you write, either do not exit the other process until the other process has a channel to the mailbox, or use a permanent mailbox.

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

Example 2–5 calculates gross income and taxes and then uses the results to calculate net income. INCOME.FOR uses SYS$CREPRC, specifying a termination mailbox, to create a subprocess to calculate taxes (CALC_TAXES) while INCOME calculates gross income. INCOME issues an asynchronous read to the termination mailbox, specifying an event flag to be set when the read completes. (The read completes when CALC_TAXES completes, terminating the created process and causing the system to write to the termination mailbox.) After finishing its own gross income calculations, INCOME.FOR waits for the flag that indicates CALC_TAXES has completed and then figures net income.

CALC_TAXES.FOR passes the tax information to INCOME.FOR using the installed common block created from INSTALLED.FOR.

Example 2–5 Asynchronous I/O Using a Mailbox

```
!INSTALLED.FOR

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

END

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

(continued on next page)

Example 2–5 (Cont.) Asynchronous I/O Using a Mailbox

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

(continued on next page)

Example 2–5 (Cont.) Asynchronous I/O Using a Mailbox

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

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
```

(continued on next page)

Example 2–5 (Cont.) Asynchronous I/O Using a Mailbox

```
! Calculate taxes
   .
   .
   .
END
```

3
Process Control

This chapter describes how to use operating system features to control a process or kernel thread. It contains the following sections:

Section 3.1 describes the control of a process or kernel thread to complete a programming task.

Section 3.2 describes how to use the operating system's process information services to gather information about a process or kernel thread.

Section 3.3 describes how to change a process's scheduling.

Section 3.4 describes the affinity and capability mechanisms for CPU scheduling.

Section 3.5 describes how to change a process's name.

Section 3.6 describes how to access another process's context.

Section 3.7 describes how to synchronize programs by setting specific times for program execution.

Section 3.8 describes how to suspend, resume, and stop program execution.

3.1 Using Process Control for Programming Tasks

Process control features in the operating system allow you to employ the following techniques to design your application:

- Modularize application programs so that each process or kernel thread of the application executes a single task
- Perform parallel processing, in which one process or kernel thread executes one part of a program while another process or kernel thread executes another part
- Implement application program control, in which one process manages and coordinates the activities of several other processes
- Schedule program execution
- Dedicate a process to execute DCL commands
- Isolate code for one or more of the following reasons:
  - To debug logic errors
  - To execute privileged code
  - To execute sensitive code

Among the services and routines the operating system provides to help you monitor and control the processes or kernel threads involved in your application are those that perform the following functions:

- Obtaining process information

- Obtaining kernel thread information
- Setting process privileges
- Setting process name
- Setting process scheduling
- Hibernating or suspending a process or kernel thread
- Deleting a process
- Synchronizing process execution

You can use system routines and DCL commands to accomplish these tasks. Table 3–1 summarizes which routines and commands to use. You can use the DCL commands in a command procedure that is executed as soon as the subprocess (or detached process) is created.

For process synchronization techniques other than specifying a time for program execution, refer to Chapter 4, Chapter 14, and Chapter 15.

Table 3–1 Routines and Commands for Controlling Processes and Kernel Threads

Routine	DCL Command	Task
LIB$GETJPI SYS$GETJPI SYS$GETJPIW	SHOW PROCESS	Return process or kernel thread information. SYS$GETJPI(W) can request process and thread information from a specific PID or PRCNAM. If no specific thread is identified, then the data represents the initial thread.
SYS$SETPRV	SET PROCESS	Set process privileges.
SYS$SETPRI	SET PROCESS	Set process or kernel thread priority. This service affects the base and current priority of a specified kernel thread and not the entire process.
SYS$SETSWM	SET PROCESS	Control swapping of process.
SYS$HIBER SYS$SUSPND SYS$RESUME	SET PROCESS	Hibernate, suspend and resume a process or kernel threads. These services hibernate, suspend, or resume all kernel threads associated with the specified process.
SYS$SETPRN	SET PROCESS	Set process name
SYS$FORCEX SYS$EXIT	EXIT and STOP	Initiate process and image rundown. All associated kernel threads of a specified process are run down and deleted.
SYS$DELPRC	EXIT and STOP	Delete process.
SYS$CANTIM	CANCEL	Cancel timer for process or kernel threads. This service finds and cancels all timers for all threads associated with the specified process.
SYS$ADJSTK	SET PROCESS	Adjust or initialize a stack pointer. Stack adjustments are performed for the kernel thread requesting the service.

(continued on next page)

Table 3–1 (Cont.) Routines and Commands for Controlling Processes and Kernel Threads

Routine	DCL Command	Task
SYS$PROCESS_SCAN	SHOW PROCESS	Scans for a process or kernel thread on the local system, or across the nodes in a VMScluster system.
SYS$SETSTK	None available	Allows the current process or kernel thread to change the size of its stacks. This service adjusts the size of the stacks of the kernel thread that invoked the service.

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

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

3.1.1 Determining Privileges for Process Creation and Control

There are three levels of process control privilege.

- Processes with the same UIC can always issue process control services for one another.
- You need the GROUP privilege to issue process control services for other processes executing in the same group.
- You need the WORLD privilege to issue process control services for any process in the system.

You need additional privileges to perform some specific functions; for example, raising the base priority of a process requires ALTPRI privilege.

3.1.2 Determining Process Identification

There are two types of process identification:

- Process identification (PID) number

  The system assigns this unique 32-bit number to a process when it is created. If you provide the **pidadr** argument to the SYS$CREPRC system service, the system returns the process identification number at the location specified. You can then use the process identification number in subsequent process control services.

- Process name

  There are two types of process names:

  - Process name

A process name is a 1- to 15-character name string. Each process name must be unique within its group (processes in different groups can have the same name). You can assign a name to a process by specifying the **prcnam** argument when you create it. You can then use this name to refer to the process in other system service calls. Note that you cannot use a process name to specify a process outside the caller's group; you must use a process identification (PID) number.

– Full process name

  The full process name is unique for each process in the cluster. Full process name strings can be up to 23 characters long and are configured in the following way:

  1–6 characters for the node name
  2 characters for the colons (::) that follow the node name
  1–15 characters for the local process name

For example, you could call the SYS$CREPRC system service, as follows:

```
unsigned int orionid=0, status;
$DESCRIPTOR(orion,"ORION");
   .
   .
   .
status = SYS$CREPRC(&orionid,           /* pidadr (process id returned) */
                    &orion,             /* prcnam  - process name */
                  . . .  );
```

The service returns the process identification in the longword at ORIONID. You can now use either the process name (ORION) or the PID (ORIONID) to refer to this process in other system service calls.

A process can set or change its own name with the Set Process Name ($SETPRN) system service. For example, a process can set its name to CYGNUS, as follows:

```
/* Descriptor for process name */
        $DESCRIPTOR(cygnus,"CYGNUS");

        status = SYS$SETPRN( &cygnus );   /* prcnam -  process name */
```

Most of the process control services accept the **prcnam** or the **pidadr** argument or both. However, you should identify a process by its process identification number for the following reasons:

- The service executes faster because it does not have to search a table of process names.
- For a process not in your group, you must use the process identification number (see Section 3.1.3).

If you specify the PID address, the service uses the PID address. If you specify the process name without a PID address, the sevice uses the process name. If you specify both—the process name and PID address—the PID address is used unless the contents of the PID is 0. In that case, the process name is used. If you specify a PID address of 0 without a process name, then the service is performed for the calling process.

If you specify neither the process name argument nor the process identification number argument, the service is performed for the calling process. If the PID address is specified, the service returns the PID of the target process in it. Table 3–2 summarizes the possible combinations of these arguments and explains how the services interpret them.

Table 3–2 Process Identification

Process Name Specified?	PID Address Specified?	Contents of PID	Resultant Action by Services
No	No	–	The process identification of the calling process is used, but is not returned.
No	Yes	0	The process identification of the calling process is used and returned.
No	Yes	PID	The process identification is used and returned.
Yes	No	–	The process name is used. The process identification is not returned.
Yes	Yes	0	The process name is used and the process identification is returned.
Yes	Yes	PID	The process identification is used and returned; the process name is ignored.

3.1.3 Qualifying Process Naming Within Groups

Process names are always qualified by their group number. The system maintains a table of all process names and the UIC associated with each. When you use the **prcnam** argument in a process control service, the table is searched for an entry that contains the specified process name and the group number of the calling process.

To use process control services on processes within its group, a calling process must have the GROUP user privilege; this privilege is not required when you specify a process with the same UIC as the caller.

The search for a process name fails if the specified process name does not have the same group number as the caller. The search fails even if the calling process has the WORLD user privilege. To execute a process control service for a process that is not in the caller's group, the requesting process must use a process identification and must have the WORLD user privilege.

3.2 Obtaining Process Information

The operating system's process information procedures enable you to gather information about processes and kernel threads. You can obtain information about one process or a group of processes on the local system or on remote nodes in a VMScluster system. You can also obtain process lock information. DCL commands such as SHOW SYSTEM and SHOW PROCESS use the process information procedures to display information about processes. You can also use the process information procedures within your programs.

The following are process information procedures:

- Get Job/Process Information (SYS$GETJPI(W))
- Get Job/Process Information (LIB$GETJPI)
- Process Scan (SYS$PROCESS_SCAN)

- Get Lock Information (SYS$GETLKI)

The SYS$GETJPI(W) and SYS$PROCESS_SCAN system services can also be used to get kernel threads information. SYS$GETJPI(W) can request threads information from a particular process ID or process name. SYS$PROCESS_SCAN can request information about all threads in a process, or all threads for each multithreaded process on the system.

For more information about SYS$GETJPI, SYS$PROCESS_SCAN, and SYS$GETLKI, see the *OpenVMS System Services Reference Manual*.

The differences among these procedures are as follows:

- SYS$GETJPI operates asynchronously.
- SYS$GETJPIW and LIB$GETJPI operate synchronously.
- SYS$GETJPI and SYS$GETJPIW can obtain one or more pieces of information about a process or kernel thread in a single call.
- LIB$GETJPI can obtain only one piece of information about a process or kernel thread in a single call.
- SYS$GETJPI and SYS$GETJPIW can specify an AST to execute at the completion of the routine.
- SYS$GETJPI and SYS$GETJPIW can use an I/O status block (IOSB) to test for completion of the routine.
- LIB$GETJPI can return some items either as strings or as numbers. It is often the easiest to call from a high-level language because the caller is not required to construct an item list.
- SYS$GETLKI returns information about the lock database.

3.2.1 Using the PID to Obtain Information

The process information procedures return information about processes by using the process identification (PID) or the process name. The PID is a 32-bit number that is unique for each process in the cluster. Specify the PID by using the **pidadr** argument. All the significant digits of a PID must be specified; only leading zeros can be omitted.

With kernel threads, the PID continues to identify a process, but can also identify a kernel thread within that process. In a multithreaded process each kernel thread has its own PID which is based on the initial threads PID.

3.2.2 Using the Process Name to Obtain Information

To obtain information about a process using the process name, specify the **prcnam** argument. Although a PID is unique for each process in the cluster, a process name is unique (within a UIC group) only for each process on a node. To locate information about processes on the local node, specify a process name string of 1 to 15 characters. To locate information about a process on a particular node, specify the full process name, which can be up to 23 characters long. The full process name is configured in the following way:

- 1 to 6 characters for the node name
- 2 characters for the colons (::) that follow the node name
- 1 to 15 characters for the local process name

Note that a local process name can look like a remote process name. Therefore, if you specify ATHENS::SMITH, the system checks for a process named ATHENS::SMITH on the local node before checking node ATHENS for a process named SMITH.

OpenVMS Programming Interfaces: Calling a System Routine and the *OpenVMS System Services Reference Manual* describe these routines completely, listing all items of information that you can request. LIB$GETJPI, SYS$GETJPI, and SYS$GETJPIW share the same item codes with the following exception: LIB$K_ items can be accessed only by LIB$GETJPI.

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

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

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

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

To specify a list of items for SYS$GETJPI or SYS$GETJPI(W) (even if that list contains only one item), use a record structure. Example 3–1 uses SYS$GETJPI(W) to request the process name and user name associated with the process whose process identification number is in SUBPROCESS_PID.

Example 3–1 Obtaining Different Types of Process Information

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

! Include the request codes
INCLUDE '($JPIDEF)'
```

(continued on next page)

Example 3–1 (Cont.) Obtaining Different Types of Process Information

```
! 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
```

(continued on next page)

Example 3–1 (Cont.) Obtaining Different Types of Process Information

```
! 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
   .
   .
   .
```

3.2.3 Using SYS$GETJPI and LIB$GETJPI

SYS$GETJPI uses the PID or the process name to obtain information about one process and the −1 wildcard as the **pidadr** to obtain information about all processes on the local system. If a PID or process name is not specified, SYS$GETJPI returns information about the calling process. SYS$GETJPI cannot perform a selective search—it can search for only one process at a time in the cluster or for all processes on the local system. If you want to perform a selective search for information or get information about processes across the cluster, use SYS$GETJPI with SYS$PROCESS_SCAN.

3.2.3.1 Requesting Information About a Single Process

Example 3–2 is a Fortran program that displays the process name and the PID of the calling program. If you want to get the same information about each process on the system, specify the initial process identification argument as −1 when you invoke LIB$GETJPI or SYS$GETJPI(W). Call the GETJPI routine (whichever you choose) repeatedly until it returns a status of SS$_NOMOREPROC, indicating that all processes on the system have been examined.

Example 3–2 Using SYS$GETJPI to Obtain Calling Process Information

```
! No process name or PID is specified; $GETJPI returns data on the
! calling process.
        PROGRAM CALLING_PROCESS

        IMPLICIT NONE                 ! Implicit none

        INCLUDE '($jpidef)   /nolist'   ! Definitions for $GETJPI

        INCLUDE '($ssdef)    /nolist'    ! System status codes

        STRUCTURE /JPIITMLST/           ! Structure declaration for
         UNION                          !  $GETJPI item lists
          MAP
           INTEGER*2 BUFLEN,
        2            CODE
           INTEGER*4 BUFADR,
        2            RETLENADR
```

(continued on next page)

Example 3–2 (Cont.) Using SYS$GETJPI to Obtain Calling Process Information

```
          END MAP
          MAP                           ! A longword of 0 terminates
           INTEGER*4 END_LIST           !  an item list
          END MAP
         END UNION
        END STRUCTURE
        RECORD /JPIITMLST/              ! Declare the item list for
        2         JPILIST(3)            !  $GETJPI

        INTEGER*4 SYS$GETJPIW           ! System service entry points

        INTEGER*4 STATUS,               ! Status variable
        2         PID                   ! PID from $GETJPI

        INTEGER*2 IOSB(4)               ! I/O Status Block for $GETJPI

        CHARACTER*16
        2         PRCNAM                ! Process name from $GETJPI
        INTEGER*2 PRCNAM_LEN            ! Process name length
        ! Initialize $GETJPI item list

        JPILIST(1).BUFLEN    = 4
        JPILIST(1).CODE      = JPI$_PID
        JPILIST(1).BUFADR    = %LOC(PID)
        JPILIST(1).RETLENADR = 0
        JPILIST(2).BUFLEN    = LEN(PRCNAM)
        JPILIST(2).CODE      = JPI$_PRCNAM
        JPILIST(2).BUFADR    = %LOC(PRCNAM)
        JPILIST(2).RETLENADR = %LOC(PRCNAM_LEN)
        JPILIST(3).END_LIST  = 0
        ! Call $GETJPI to get data for this process

        STATUS = SYS$GETJPIW (
        2                    ,           ! No event flag
        2                    ,           ! No PID
        2                    ,           ! No process name
        2                    JPILIST,    ! Item list
        2                    IOSB,       ! Always use IOSB with $GETJPI!
        2                    ,           ! No AST
        2                    )           ! No AST arg
        ! Check the status in both STATUS and the IOSB, if
        ! STATUS is OK then copy IOSB(1) to STATUS

        IF (STATUS) STATUS = IOSB(1)

        ! If $GETJPI worked, display the process, if done then
        ! prepare to exit, otherwise signal an error

        IF (STATUS) THEN
                TYPE 1010, PID, PRCNAM(1:PRCNAM_LEN)
1010                FORMAT (' ',Z8.8,'  ',A)
        ELSE
                CALL LIB$SIGNAL(%VAL(STATUS))
        END IF

        END
```

Example 3–3 creates the file PROCNAME.RPT that lists, using LIB$GETJPI, the process name of each process on the system. If the process running this program does not have the privilege necessary to access a particular process, the program writes the words NO PRIVILEGE in place of the process name. If a process is suspended, LIB$GETJPI cannot access it and the program writes the word SUSPENDED in place of the process name. Note that, in either of these cases, the program changes the error value in STATUS to a success value so that the loop calling LIB$GETJPI continues to execute.

Example 3–3 Obtaining the Process Name

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

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

! Process name
CHARACTER*15 NAME
INTEGER LEN
! Process identification
INTEGER PID /-1/
   .
   .
   .
! Get logical unit number and open the file
STATUS = LIB$GET_LUN (LUN)
OPEN (UNIT = LUN,
2     FILE = 'PROCNAME.RPT',
2     STATUS = 'NEW')
! Get information and write it to file
DO WHILE (STATUS)
  STATUS = LIB$GETJPI(JPI$_PRCNAM,
2                     PID,
2                     ,,
2                     NAME,
2                     LEN)
```

(continued on next page)

Example 3–3 (Cont.) Obtaining the Process Name

```
  ! 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)
   .
   .
   .
```

Example 3–4 demonstrates how to use the process name to obtain information about a process.

Example 3–4 Using SYS$GETJPI and the Process Name to Obtain Information About a Process

```
! To find information for a particular process by name,
! substitute this code, which includes a process name,
! to call $GETJPI in Example 3-2

! Call $GETJPI to get data for a named process

STATUS = SYS$GETJPIW (
2                    ,           ! No event flag
2                    ,           ! No PID
2                    'SMITH_1',  ! Process name
2                    JPILIST,    ! Item list
2                    IOSB,       ! Always use IOSB with $GETJPI!
2                    ,           ! No AST
2                    )           ! No AST arg
```

3.2.3.2 Requesting Information About All Processes on the Local System

You can use SYS$GETJPI to perform a wildcard search on all processes on the local system. When the initial **pidadr** argument is specified as –1, SYS$GETJPI returns requested information for each process that the program has privilege to access. The requested information is returned for one process per call to SYS$GETJPI.

To perform a wildcard search, call SYS$GETJPI in a loop, testing the return status.

When performing wildcard searches, SYS$GETJPI returns an error status for processes that are inaccessible. When a program that uses a −1 wildcard checks the status value returned by SYS$GETJPI, it should test for the following status codes:

Status	Explanation
SS$_NOMOREPROC	All processes have been returned.
SS$_NOPRIV	The caller lacks sufficient privilege to examine a process.
SS$_SUSPENDED	The target process is being deleted or is suspended and cannot return the information.

Example 3–5 is a C program that demonstrates how to use the SYS$GETJPI −1 wildcard to search for all processes on the local system.

Example 3–5 Using SYS$GETJPI to Request Information About All Processes on the Local System

```
#include <stdio.h>
#include <jpidef.h>
#include <stdlib.h>
#include <ssdef.h>

/* Item descriptor */

struct {
        unsigned short buflen, item_code;
        void *bufaddr;
        void *retlenaddr;
        unsigned int terminator;
}itm_lst;

/* I/O Status Block */

struct {
        unsigned short iostat;
        unsigned short iolen;
        unsigned int device_info;
}iosb;

main() {

        unsigned short len;
        unsigned int efn=1,pidadr = -1,status, usersize;
        char username[12];

/* Initialize the item list */

        itm_lst.buflen = 12;
        itm_lst.item_code = JPI$_USERNAME;
        itm_lst.bufaddr = username;
        itm_lst.retlenaddr = &usersize;
        itm_lst.terminator = 0;

        do{
```

(continued on next page)

Example 3–5 (Cont.) Using SYS$GETJPI to Request Information About All Processes on the Local System

```
        status = SYS$GETJPIW(0,                  /* no event flag */
                             &pidadr,            /* process id */
                             0,                  /* process name */
                             &itm_lst,           /* item list */
                             &iosb,              /* I/O status block */
                             0,                  /* astadr (AST routine) */
                             0);                 /* astprm (AST parameter) */
                 switch(status)
                 {
case SS$_NOPRIV:
                printf("\nError: No privileges for attempted operation");
                break;
case SS$_SUSPENDED:
                printf("\nError: Process is suspended");
                break;
case SS$_NORMAL:
                if (iosb.iostat == SS$_NORMAL)
                     printf("\nUsername: %s",username);
                else
                     printf("\nIOSB condition value  %d returned",iosb.iostat);
                        }

        }while(status != SS$_NOMOREPROC);

}
```

3.2.4 Using SYS$GETJPI with SYS$PROCESS_SCAN

Using the SYS$PROCESS_SCAN system service greatly enhances the power of SYS$GETJPI. With this combination, you can search for selected groups of processes or kernel threads on the local system as well as for processes or kernel threads on remote nodes or across the cluster. When you use SYS$GETJPI alone, you specify the **pidadr** or the **prcnam** argument to locate information about one process. When you use SYS$GETJPI with SYS$PROCESS_SCAN, the **pidctx** argument generated by SYS$PROCESS_SCAN is used as the **pidadr** argument to SYS$GETJPI. This context allows SYS$GETJPI to use the selection criteria set up in the call to SYS$PROCESS_SCAN.

When using SYS$GETJPI with a PRCNAM specified, SYS$GETJPI returns data for only the initial thread. This parallels the behavior of the DCL commands SHOW SYSTEM, SHOW PROCESS, and MONITOR PROCESS. If a valid PIDADR is specified, then the data returned describes only that specific kernel thread. If a PIDADR of zero is specified, then the data returned describes the calling kernel thread.

SYS$GETJPI has the flag, JPI$_THREAD, as part of the JPI$_GETJPI_CONTROL_FLAGS item code. The JPI$_THREAD flag designates that the service call is requesting data for all of the kernel threads in a multithreaded process. If the call is made with JPI$_THREAD set, then SYS$GETJPI begins with the initial thread, and SYS$GETJPI returns SS$_NORMAL. Subsequent calls to SYS$GETJPI with JPI$_THREAD specified returns data for the next thread until there are no more threads, at which time the service returns SS$_NOMORETHREAD.

If you specify a wildcard PIDADR −1 along with JPI$_THREAD, you cause SYS$GETJPI to return information for all threads for all processes on the system on subsequent calls. SYS$GETJPI returns the status SS$_NORMAL until there are no more processes, at which time it returns SS$_NOMOREPROC. If you specify a wildcard search, you must request either the JPI$_PROC_INDEX or the JPI$_INITIAL_THREAD_PID item code to distinguish the transition from the last thread of a multithreaded process to the next process. The PROC_INDEX and the INITIAL_THREAD_PID are different for each process on the system.

Table 3–3 shows four item codes of SYS$GETJPI that provide kernel thread information.

Table 3–3 SYS$GETJPI Kernel Threads Item Codes

Item Code	Meaning
JPI$_INITIAL_ THREAD_PID	Returns the PID of the initial thread for the target process.
JPI$_KT_COUNT	Returns the current count of kernel threads for the target process.
JPI$_MULTITHREAD	Returns the maximum kernel thread count allowed for the target process.
JPI$_THREAD_ INDEX	Returns the kernel thread index for the target thread or process.

This wildcard search is initiated by invoking SYS$GETJPI with a −1 specified for the PID, and is only available on the local node. With kernel threads, a search for all threads in a single process is available, both on the local node and on another node on the cluster.

In a dual architecture or mixed-version cluster, it is possible that one or more nodes in the cluster do not support kernel threads. To indicate this condition, a threads capability bit (CSB$M_CAP_THREADS) exists in the CSB$L_ CAPABILITY cell in the cluster status block. If this bit is set for a node, it indicates that the node supports kernel threads. This information is passed around as part of the normal cluster management activity when a node is added to a cluster. If a SYS$GETJPI request that requires thread support needs to be passed to another node in the cluster, a check is made on whether the node supports kernel threads before the request is sent to that node. If the node supports kernel threads, the request is sent. If the node does not support kernel threads, the status SS$_INCOMPAT is returned to the caller and the request is not sent to the other node.

You can use SYS$PROCESS_SCAN only with SYS$GETJPI; you cannot use it alone. The process context generated by SYS$PROCESS_SCAN is used like the −1 wildcard except that it is initialized by calling the SYS$PROCESS_ SCAN service instead of by a simple assignment statement. However, the SYS$PROCESS_SCAN context is more powerful and more flexible than the −1 wildcard. SYS$PROCESS_SCAN uses an item list to specify selection criteria to be used in a search for processes and produces a context longword that describes a selective search for SYS$GETJPI.

Using SYS$GETJPI with SYS$PROCESS_SCAN to perform a selective search is a more efficient way to locate information because only information about the processes you have selected is returned. For example, you can specify a search for processes owned by one user name, and SYS$GETJPI returns only the processes that match the specified user name. You can specify a search for all batch processes and SYS$GETJPI returns only information about processes running as batch jobs. You can specify a search for all batch processes owned by one user name and SYS$GETJPI returns only information about processes owned by that user name that are running as batch jobs.

By default, SYS$PROCESS_SCAN sets up a context for only the initial thread of a multithreaded process. However, if the value PSCAN$_THREAD is specified for the item code PSCAN$_PSCAN_CONTOL_FLAGS, then threads are included in the scan. The PSCAN$_THREAD flag takes precedence over the JPI$_THREAD flag in the SYS$GETJPI call. With PSCAN$_THREAD specified, threads are included in the entire scan. With PSCAN$_THREAD not specified, threads are included in the scan for a specific SYS$GETJPI call only if JPI$_THREAD is specified.

Table 3–4 shows two item codes of SYS$PROCESS_SCAN that provide kernel thread information.

Table 3–4 SYS$PROCESS_SCAN Kernel Threads Item Codes

Item Code	Meaning
PSCAN$_KT_COUNT	Uses the current count of kernel threads for the process as a selection criteria. The valid item-specific flags for this item code are EQL, GEQ, GTR, LEQ, LSS, NEQ, and OR.
PSCAN$_ MULTITHREAD	Uses the maximum count of kernel threads for the process as a selection criteria. The valid item-specific flags for this item codes are EQL, GEQ, GTR, LEQ, LSS, NEQ, and OR.

3.2.4.1 Using SYS$PROCESS_SCAN Item List and Item-Specific Flags

SYS$PROCESS_SCAN uses an item list to specify the selection criteria for the SYS$GETJPI search.

Each entry in the SYS$PROCESS_SCAN item list contains the following:

- The attribute of the process to be examined
- The value of the attribute or a pointer to the value
- Item-specific flags to control how to interpret the value

Item-specific flags enable you to control selection information. For example, you can use flags to select only those processes that have attribute values that correspond to the value in the item list, as shown in Table 3–5.

Table 3–5 Item-Specific Flags

Item-Specific Flag	Description
PSCAN$M_OR	Match this value or the next value.
PSCAN$M_EQL	Match value exactly (the default.)
PSCAN$M_NEQ	Match if value is not equal.
PSCAN$M_GEQ	Match if value is greater than or equal to.
PSCAN$M_GTR	Match if value is greater than.

(continued on next page)

Table 3–5 (Cont.) Item-Specific Flags

Item-Specific Flag	Description
PSCAN$M_LEQ	Match if value is less than or equal to.
PSCAN$M_LSS	Match if value is less than.
PSCAN$M_CASE_BLIND	Match without regard to case of letters.
PSCAN$M_PREFIX_MATCH	Match on the leading substring.
PSCAN$M_WILDCARD	Match string is a wildcard pattern.

The PSCAN$M_OR flag is used to connect entries in an item list. For example, in a program that searches for processes owned by several specified users, each user name must be specified in a separate item list entry. The item list entries are connected with the PSCAN$M_OR flag as shown in the following Fortran example. This example connects all the processes on the local node that belong to SMITH, JONES, or JOHNSON.

```
PSCANLIST(1).BUFLEN   = LEN('SMITH')
PSCANLIST(1).CODE     = PSCAN$_USERNAME
PSCANLIST(1).BUFADR   = %LOC('SMITH')
PSCANLIST(1).ITMFLAGS = PSCAN$M_OR
PSCANLIST(2).BUFLEN   = LEN('JONES')
PSCANLIST(2).CODE     = PSCAN$_USERNAME
PSCANLIST(2).BUFADR   = %LOC('JONES')
PSCANLIST(2).ITMFLAGS = PSCAN$M_OR
PSCANLIST(3).BUFLEN   = LEN('JOHNSON')
PSCANLIST(3).CODE     = PSCAN$_USERNAME
PSCANLIST(3).BUFADR   = %LOC('JOHNSON')
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).END_LIST = 0
```

Use the PSCAN$M_WILDCARD flag to specify that a character string is to be treated as a wildcard. For example, to find all process names that begin with the letter A and end with the string ER, use the string A*ER with the PSCAN$M_WILDCARD flag. If the PSCAN$M_WILDCARD flag is not specified, the search looks for the 4-character process name A*ER.

The PSCAN$M_PREFIX_MATCH defines a wildcard search to match the initial characters of a string. For example, to find all process names that start with the letters AB, use the string AB with the PSCAN$M_PREFIX_MATCH flag. If you do not specify the PSCAN$M_PREFIX_MATCH flag, the search looks for a process with the 2-character process name AB.

3.2.4.2 Requesting Information About Processes That Match One Criterion

You can use SYS$GETJPI with SYS$PROCESS_SCAN to search for processes that match an item list with one criterion. For example, if you specify a search for processes owned by one user name, SYS$GETJPI returns only those processes that match the specified user name.

Example 3–6 demonstrates how to perform a SYS$PROCESS_SCAN search on the local node to select all processes that are owned by user SMITH.

Example 3–6 Using SYS$GETJPI and SYS$PROCESS_SCAN to Select Process Information by User Name

```
PROGRAM PROCESS_SCAN

IMPLICIT NONE                   ! Implicit none

INCLUDE '($jpidef)   /nolist'   ! Definitions for $GETJPI
INCLUDE '($pscandef) /nolist'   ! Definitions for $PROCESS_SCAN
INCLUDE '($ssdef)    /nolist'   ! Definitions for SS$_NAMES

STRUCTURE /JPIITMLST/           ! Structure declaration for
 UNION                          !  $GETJPI item lists
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            RETLENADR
  END MAP
  MAP                           ! A longword of 0 terminates
   INTEGER*4 END_LIST           !  an item list
  END MAP
 END UNION
END STRUCTURE
STRUCTURE /PSCANITMLST/         ! Structure declaration for
 UNION                          !  $PROCESS_SCAN item lists
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            ITMFLAGS
  END MAP
  MAP                           ! A longword of 0 terminates
   INTEGER*4 END_LIST           !  an item list
  END MAP
 END UNION
END STRUCTURE
RECORD /PSCANITMLST/            ! Declare the item list for
2         PSCANLIST(12)         !  $PROCESS_SCAN

RECORD /JPIITMLST/              ! Declare the item list for
2         JPILIST(3)            !  $GETJPI

INTEGER*4 SYS$GETJPIW,          ! System service entry points
2         SYS$PROCESS_SCAN

INTEGER*4 STATUS,               ! Status variable
2         CONTEXT,              ! Context from $PROCESS_SCAN
2         PID                   ! PID from $GETJPI

INTEGER*2 IOSB(4)               ! I/O Status Block for $GETJPI

CHARACTER*16
2         PRCNAM                ! Process name from $GETJPI
INTEGER*2 PRCNAM_LEN            ! Process name length

LOGICAL*4 DONE                  ! Done with data loop
```

(continued on next page)

Example 3–6 (Cont.) Using SYS$GETJPI and SYS$PROCESS_SCAN to Select Process Information by User Name

```
        !********************************************
        !*  Initialize item list for $PROCESS_SCAN  *
        !********************************************

        ! Look for processes owned by user SMITH

        PSCANLIST(1).BUFLEN   = LEN('SMITH')
        PSCANLIST(1).CODE     = PSCAN$_USERNAME
        PSCANLIST(1).BUFADR   = %LOC('SMITH')
        PSCANLIST(1).ITMFLAGS = 0
        PSCANLIST(2).END_LIST = 0
        !********************************************
        !*     End of item list initialization      *
        !********************************************

        STATUS = SYS$PROCESS_SCAN (        ! Set up the scan context
        2                         CONTEXT,
        2                         PSCANLIST)

        IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

        ! Loop calling $GETJPI with the context

        DONE = .FALSE.
        DO WHILE (.NOT. DONE)

                ! Initialize $GETJPI item list

                JPILIST(1).BUFLEN    = 4
                JPILIST(1).CODE      = JPI$_PID
                JPILIST(1).BUFADR    = %LOC(PID)
                JPILIST(1).RETLENADR = 0
                JPILIST(2).BUFLEN    = LEN(PRCNAM)
                JPILIST(2).CODE      = JPI$_PRCNAM
                JPILIST(2).BUFADR    = %LOC(PRCNAM)
                JPILIST(2).RETLENADR = %LOC(PRCNAM_LEN)
                JPILIST(3).END_LIST  = 0
        ! Call $GETJPI to get the next SMITH process

                STATUS = SYS$GETJPIW (
        2                     ,            ! No event flag
        2                     CONTEXT,     ! Process context
        2                     ,            ! No process name
        2                     JPILIST,     ! Item list
        2                     IOSB,        ! Always use IOSB with $GETJPI!
        2                     ,            ! No AST
        2                     )            ! No AST arg
                ! Check the status in both STATUS and the IOSB, if
                ! STATUS is OK then copy IOSB(1) to STATUS

                IF (STATUS) STATUS = IOSB(1)

                ! If $GETJPI worked, display the process, if done then
                ! prepare to exit, otherwise signal an error

                IF (STATUS) THEN
                        TYPE 1010, PID, PRCNAM(1:PRCNAM_LEN)
1010                        FORMAT (' ',Z8.8,'  ',A)
                ELSE IF (STATUS .EQ. SS$_NOMOREPROC) THEN
                        DONE = .TRUE.
                ELSE
                        CALL LIB$SIGNAL(%VAL(STATUS))
                END IF
```

(continued on next page)

Example 3–6 (Cont.) Using SYS$GETJPI and SYS$PROCESS_SCAN to Select Process Information by User Name

```
END DO

END
```

3.2.4.3 Requesting Information About Processes That Match Multiple Values for One Criterion

You can use SYS$PROCESS_SCAN to search for processes that match one of a number of values for a single criterion, such as processes owned by several specified users.

Each value must be specified in a separate item list entry, and the item list entries must be connected with the PSCAN$M_OR item-specific flag. SYS$GETJPI selects each process that matches any of the item values.

For example, to look for processes with user names SMITH, JONES, or JOHNSON, substitute code such as that shown in Example 3–7 to initialize the item list in Example 3–6.

Example 3–7 Using SYS$GETJPI and SYS$PROCESS_SCAN with Multiple Values for One Criterion

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Look for users SMITH, JONES and JOHNSON

PSCANLIST(1).BUFLEN   = LEN('SMITH')
PSCANLIST(1).CODE     = PSCAN$_USERNAME
PSCANLIST(1).BUFADR   = %LOC('SMITH')
PSCANLIST(1).ITMFLAGS = PSCAN$M_OR
PSCANLIST(2).BUFLEN   = LEN('JONES')
PSCANLIST(2).CODE     = PSCAN$_USERNAME
PSCANLIST(2).BUFADR   = %LOC('JONES')
PSCANLIST(2).ITMFLAGS = PSCAN$M_OR
PSCANLIST(3).BUFLEN   = LEN('JOHNSON')
PSCANLIST(3).CODE     = PSCAN$_USERNAME
PSCANLIST(3).BUFADR   = %LOC('JOHNSON')
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).END_LIST = 0

!********************************************
!*     End of item list initialization      *
!********************************************
```

3.2.4.4 Requesting Information About Processes That Match Multiple Criteria

You can use SYS$PROCESS_SCAN to search for processes that match values for more than one criterion. When multiple criteria are used, a process must match at least one value for each specified criterion.

Example 3–8 demonstrates how to find any batch process owned by either SMITH or JONES. The program uses syntax like the following logical expression to initialize the item list:

```
((username = "SMITH") OR (username = "JONES"))
                    AND
            (MODE = JPI$K_BATCH)
```

Example 3–8 Selecting Processes That Match Multiple Criteria

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Look for BATCH jobs owned by users SMITH and JONES

PSCANLIST(1).BUFLEN   = LEN('SMITH')
PSCANLIST(1).CODE     = PSCAN$_USERNAME
PSCANLIST(1).BUFADR   = %LOC('SMITH')
PSCANLIST(1).ITMFLAGS = PSCAN$M_OR
PSCANLIST(2).BUFLEN   = LEN('JONES')
PSCANLIST(2).CODE     = PSCAN$_USERNAME
PSCANLIST(2).BUFADR   = %LOC('JONES')
PSCANLIST(2).ITMFLAGS = 0
PSCANLIST(3).BUFLEN   = 0
PSCANLIST(3).CODE     = PSCAN$_MODE
PSCANLIST(3).BUFADR   = JPI$K_BATCH
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).END_LIST = 0

!********************************************
!*     End of item list initialization      *
!********************************************
```

See the *OpenVMS System Services Reference Manual* for more information about SYS$PROCESS_SCAN item codes and flags.

3.2.5 Specifying a Node as Selection Criterion

Several SYS$PROCESS_SCAN item codes do not refer to attributes of a process, but to the VMScluster node on which the target process resides. When SYS$PROCESS_SCAN encounters an item code that refers to a node attribute, it creates an alphabetized list of node names. SYS$PROCESS_SCAN then directs SYS$GETJPI to compare the selection criteria against processes on these nodes.

SYS$PROCESS_SCAN ignores a node specification if it is running on a node that is not part of a VMScluster system. For example, if you request that SYS$PROCESS_SCAN select all nodes with the hardware model name VAX 6360, this search returns information about local processes on a nonclustered system, even if it is a MicroVAX system.

A remote SYS$GETJPI operation currently requires the system to send a message to the CLUSTER_SERVER process on the remote node. The CLUSTER_SERVER process then collects the information and returns it to the requesting node. This has several implications for clusterwide searches:

- All remote SYS$GETJPI operations are asynchronous and must be synchronized properly. Many applications that are not correctly synchronized might seem to work on a single node because some SYS$GETJPI operations are actually synchronous; however, these applications fail if they attempt to examine processes on remote nodes. For more information on how to synchronize SYS$GETJPI operations, see Chapter 14.
- The CLUSTER_SERVER process is always a current process, because it is executing on behalf of SYS$GETJPI.

- Attempts by SYS$GETJPI to examine a node do not succeed during a brief period between the time a node joins the cluster and the time that the CLUSTER_SERVER process is started. Searches that occur during this period skip such a node. Searches that specify only such a booting node fail with a SYS$GETJPI status of SS$_UNREACHABLE.
- SS$_NOMOREPROC is returned after all processes on all specified nodes have been scanned.

3.2.5.1 Checking All Nodes on the Cluster for Processes

The SYS$PROCESS_SCAN system service can scan the entire cluster for processes. For example, to scan the cluster for all processes owned by SMITH, use code like that in Example 3–9 to initialize the item list to find all processes with a nonzero cluster system identifier (CSID) and a user name of SMITH.

Example 3–9 Searching the Cluster for Process Information

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Search the cluster for jobs owned by SMITH

PSCANLIST(1).BUFLEN   = 0
PSCANLIST(1).CODE     = PSCAN$_NODE_CSID
PSCANLIST(1).BUFADR   = 0
PSCANLIST(1).ITMFLAGS = PSCAN$M_NEQ
PSCANLIST(2).BUFLEN   = LEN('SMITH')
PSCANLIST(2).CODE     = PSCAN$_USERNAME
PSCANLIST(2).BUFADR   = %LOC('SMITH')
PSCANLIST(2).ITMFLAGS = 0
PSCANLIST(3).END_LIST = 0

!********************************************
!*     End of item list initialization      *
!********************************************
```

3.2.5.2 Checking Specific Nodes on the Cluster for Processes

You can specify a list of nodes as well. Example 3–10 demonstrates how to design an item list to search for batch processes on node TIGNES, VALTHO, or 2ALPES.

Example 3–10 Searching for Process Information on Specific Nodes in the Cluster

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Search for BATCH jobs on nodes TIGNES, VALTHO and 2ALPES
```

(continued on next page)

Example 3–10 (Cont.) Searching for Process Information on Specific Nodes in the Cluster

```
PSCANLIST(1).BUFLEN   = LEN('TIGNES')
PSCANLIST(1).CODE     = PSCAN$_NODENAME
PSCANLIST(1).BUFADR   = %LOC('TIGNES')
PSCANLIST(1).ITMFLAGS = PSCAN$M_OR
PSCANLIST(2).BUFLEN   = LEN('VALTHO')
PSCANLIST(2).CODE     = PSCAN$_NODENAME
PSCANLIST(2).BUFADR   = %LOC('VALTHO')
PSCANLIST(2).ITMFLAGS = PSCAN$M_OR
PSCANLIST(3).BUFLEN   = LEN('2ALPES')
PSCANLIST(3).CODE     = PSCAN$_NODENAME
PSCANLIST(3).BUFADR   = %LOC('2ALPES')
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).BUFLEN   = 0
PSCANLIST(4).CODE     = PSCAN$_MODE
PSCANLIST(4).BUFADR   = JPI$K_BATCH
PSCANLIST(4).ITMFLAGS = 0
PSCANLIST(5).END_LIST = 0

!********************************************
!*     End of item list initialization      *
!********************************************
```

3.2.5.3 Conducting Multiple Simultaneous Searches with SYS$PROCESS_SCAN

Only one asynchronous remote SYS$GETJPI request per SYS$PROCESS_SCAN context is permitted at a time. If you issue a second SYS$GETJPI request using a context before a previous remote request using the same context has completed, your process stalls in a resource wait until the previous remote SYS$GETJPI request completes. This stall in the RWAST state prevents your process from executing in user mode or receiving user-mode ASTs.

If you want to run remote searches in parallel, create multiple contexts by calling SYS$PROCESS_SCAN once for each context. For example, you can design a program that calls SYS$GETSYI in a loop to find the nodes in the VMScluster system and creates a separate SYS$PROCESS_SCAN context for each remote node. Each of these separate contexts can run in parallel. The DCL command SHOW USERS uses this technique to obtain user information more quickly.

Only requests to remote nodes must wait until the previous search using the same context has completed. If the SYS$PROCESS_SCAN context specifies the local node, any number of SYS$GETJPI requests using that context can be executed in parallel (within the limits implied by the process quotas for ASTLM and BYTLM).

Note

When you use SYS$GETJPI to reference remote processes, you must properly synchronize all SYS$GETJPI calls. Before the operating system's Version 5.2, if you did not follow these synchronization rules, your programs might have appeared to run correctly. However, if you attempt to run such improperly synchronized programs using SYS$GETJPI with SYS$PROCESS_SCAN with a remote process, your program might attempt to use the data before SYS$GETJPI has returned it.

To perform a synchronous search in which the program waits until all requested information is available, use SYS$GETJPIW with an **iosb** argument.

See the *OpenVMS System Services Reference Manual* for more information about process identification, SYS$GETJPI, and SYS$PROCESS_SCAN.

3.2.6 Programming with SYS$GETJPI

The following sections describe some important considerations for programming with SYS$GETJPI.

3.2.6.1 Using Item Lists Correctly

When SYS$GETJPI collects data, it makes multiple passes through the item list. If the item list is self-modifying—that is, if the addresses for the output buffers in the item list point back at the item list—SYS$GETJPI replaces the item list information with the returned data. Therefore, incorrect data might be read or unexpected errors might occur when SYS$GETJPI reads the item list again. To prevent confusing errors, Digital recommends that you do not use self-modifying item lists.

The number of passes that SYS$GETJPI needs depends on which item codes are referenced and the state of the target process. A program that appears to work normally might fail when a system has processes that are swapped out of memory, or when a process is on a remote node.

3.2.6.2 Improving Performance by Using Buffered $GETJPI Operations

To request information about a process located on a remote node, SYS$GETJPI must send a message to the remote node, wait for the response, and then extract the data from the message received. When you perform a search on a remote system, the program must repeat this sequence for each process that SYS$GETJPI locates.

To reduce the overhead of such a remote search, use SYS$PROCESS_SCAN with the PSCAN$_GETJPI_BUFFER_SIZE item code to specify a buffer size for SYS$GETJPI. When the buffer size is specified by SYS$PROCESS_SCAN, SYS$GETJPI packs information for several processes into one buffer and transmits them in a single message. This reduction in the number of messages improves performance.

For example, if the SYS$GETJPI item list requests 100 bytes of information, you might specify a PSCAN$_GETJPI_BUFFER_SIZE of 1000 bytes so that the service can place information for at least 10 processes in each message. (SYS$GETJPI does not send fill data in the message buffer; therefore, information for more than 10 processes can be packed into the buffer.)

The SYS$GETJPI buffer must be large enough to hold the data for at least one process. If the buffer is too small, the error code SS$_IVBUFLEN is returned from the SYS$GETJPI call.

You do not have to allocate space for the SYS$GETJPI buffer; buffer space is allocated by SYS$PROCESS_SCAN as part of the search context that it creates. Because SYS$GETJPI buffering is transparent to the program that calls SYS$GETJPI, you do not have to modify the loop that calls SYS$GETJPI.

If you use PSCAN$_GETJPI_BUFFER_SIZE with SYS$PROCESS_SCAN, all calls to SYS$GETJPI using that context must request the same item code information. Because SYS$GETJPI collects information for more than one process at a time within its buffers, you cannot change the item codes or the lengths of the buffers in the SYS$GETJPI item list between calls. SYS$GETJPI returns the error SS$_BADPARAM if any item code or buffer length changes between SYS$GETJPI calls. However, you can change the buffer addresses in the SYS$GETJPI item list from call to call.

The SYS$GETJPI buffered operation is not used for searching the local node. When a search specifies both multiple nodes and SYS$GETJPI buffering, the buffering is used on remote nodes but is ignored on the local node. Example 3–11 demonstrates how to use a SYS$GETJPI buffer to improve performance.

Example 3–11 Using a SYS$GETJPI Buffer to Improve Performance

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Search for jobs owned by users SMITH and JONES
! across the cluster with $GETJPI buffering

PSCANLIST(1).BUFLEN   = 0
PSCANLIST(1).CODE     = PSCAN$_NODE_CSID
PSCANLIST(1).BUFADR   = 0
PSCANLIST(1).ITMFLAGS = PSCAN$M_NEQ
PSCANLIST(2).BUFLEN   = LEN('SMITH')
PSCANLIST(2).CODE     = PSCAN$_USERNAME
PSCANLIST(2).BUFADR   = %LOC('SMITH')
PSCANLIST(2).ITMFLAGS = PSCAN$M_OR
PSCANLIST(3).BUFLEN   = LEN('JONES')
PSCANLIST(3).CODE     = PSCAN$_USERNAME
PSCANLIST(3).BUFADR   = %LOC('JONES')
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).BUFLEN   = 0
PSCANLIST(4).CODE     = PSCAN$_GETJPI_BUFFER_SIZE
PSCANLIST(4).BUFADR   = 1000
PSCANLIST(4).ITMFLAGS = 0
PSCANLIST(5).END_LIST = 0

!********************************************
!*     End of item list initialization     *
!********************************************
```

3.2.6.3 Fulfilling Remote SYS$GETJPI Quota Requirements

A remote SYS$GETJPI request uses system dynamic memory for messages. System dynamic memory uses the process quota BYTLM. Follow these steps to determine the number of bytes required by a SYS$GETJPI request:

1. Add the following together:
   - The size of the SYS$PROCESS_SCAN item list
   - The total size of all reference buffers for SYS$PROCESS_SCAN (the sum of all buffer length fields in the item list)
   - The size of the SYS$GETJPI item list
   - The size of the SYS$GETJPI buffer
   - The size of the calling process RIGHTSLIST
   - Approximately 300 bytes for message overhead
2. Double this total.

   The total is doubled because the messages consume system dynamic memory on both the sending node and the receiving node.

This formula for BYTLM quota applies to both buffered and nonbuffered SYS$GETJPI requests. For buffered requests, use the value specified in the SYS$PROCESS_SCAN item, PSCAN$_GETJPI_BUFFER_SIZE, as the size of the buffer. For nonbuffered requests, use the total length of all data buffers specified in the SYS$GETJPI item list as the size of the buffer.

If the BYTLM quota is insufficient, SYS$GETJPI (not SYS$PROCESS_SCAN) returns the error SS$_EXBYTLM.

3.2.6.4 Using the SYS$GETJPI Control Flags

The JPI$_GETJPI_CONTROL_FLAGS item code, which is specified in the SYS$GETJPI item list, provides additional control over SYS$GETJPI. Therefore, SYS$GETJPI may be unable to retrieve all the data requested in an item list because JPI$_GETJPI_CONTROL_FLAGS requests that SYS$GETJPI not perform certain actions that may be necessary to collect the data. For example, a SYS$GETJPI control flag may instruct the calling program not to retrieve a process that has been swapped out of the balance set.

If SYS$GETJPI is unable to retrieve any data item because of the restrictions imposed by the control flags, it returns the data length as 0. To verify that SYS$GETJPI received a data item, examine the data length to be sure that it is not 0. To make this verification possible, be sure to specify the return length for each item in the SYS$GETJPI item list when any of the JPI$_GETJPI_CONTROL_FLAGS flags is used.

Unlike other SYS$GETJPI item codes, the JPI$_GETJPI_CONTROL_FLAGS item is an input item. The item list entry should specify a longword buffer. The desired control flags should be set in this buffer.

Because the JPI$_GETJPI_CONTROL_FLAGS item code tells SYS$GETJPI how to interpret the item list, it must be the first entry in the SYS$GETJPI item list. The error code SS$_BADPARAM is returned if it is not the first item in the list.

The following are the SYS$GETJPI control flags.

JPI$M_NO_TARGET_INSWAP

When you specify JPI$M_NO_TARGET_INSWAP, SYS$GETJPI does not retrieve a process that has been swapped out of the balance set. Use JPI$M_NO_TARGET_INSWAP to avoid the additional load of swapping processes into a system. For example, use this flag with SHOW SYSTEM to avoid bringing processes into memory to display their accumulated CPU time.

If you specify JPI$M_NO_TARGET_INSWAP and request information from a process that has been swapped out, the following consequences occur:

- Data stored in the virtual address space of the process is not accessible.
- Data stored in the process header (PHD) may not be accessible.
- Data stored in resident data structures, such as the process control block (PCB) or the job information block (JIB), is accessible.

You must examine the return length of an item to verify that the item was retrieved. The information may be located in a different data structure in another release of the operating system.

JPI$M_NO_TARGET_AST

When JPI$M_NO_TARGET_AST is specified, SYS$GETJPI does not deliver a kernel-mode AST to the target process. JPI$M_NO_TARGET_AST is used to avoid executing a target process in order to retrieve information.

If you specify JPI$M_NO_TARGET_AST and cannot deliver an AST to a target process, the following consequences occur:

- Data stored in the virtual address space of the process is not accessible.
- Data stored in system data structures, such as the process header (PHD), the process control block (PCB), or the job information block (JIB), is accessible.

You must examine the return length of an item to verify that the item was retrieved. The information may be located in a different data structure in another release of the operating system.

The use of the flag JPI$M_NO_TARGET_AST also implies that SYS$GETJPI does not swap in a process, because SYS$GETJPI would only bring a process into memory to deliver an AST to that process.

JPI$M_IGNORE_TARGET_STATUS

When JPI$M_IGNORE_TARGET_STATUS is specified, SYS$GETJPI attempts to retrieve as much information as possible, even if the process is suspended or being deleted. JPI$M_IGNORE_TARGET_STATUS is used to retrieve all possible information from a process. For example, this flag is used with SHOW SYSTEM to display processes that are suspended, being deleted, or in miscellaneous wait states.

Example 3–12 demonstrates how to use SYS$GETJPI control flags to avoid swapping processes during a SYS$GETJPI call.

Example 3–12 Using SYS$GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
PROGRAM CONTROL_FLAGS

IMPLICIT NONE                   ! Implicit none

INCLUDE '($jpidef)   /nolist'   ! Definitions for $GETJPI
INCLUDE '($pscandef) /nolist'   ! Definitions for $PROCESS_SCAN
INCLUDE '($ssdef)    /nolist'   ! Definitions for SS$_ names
```

(continued on next page)

Example 3–12 (Cont.) Using SYS$GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
STRUCTURE /JPIITMLST/            ! Structure declaration for
 UNION                           !  $GETJPI item lists
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            RETLENADR
  END MAP
  MAP                            ! A longword of 0 terminates
   INTEGER*4 END_LIST            !  an item list
  END MAP
 END UNION
END STRUCTURE
STRUCTURE /PSCANITMLST/          ! Structure declaration for
 UNION                           !  $PROCESS_SCAN item lists
  MAP
   INTEGER*2 BUFLEN,
2            CODE
   INTEGER*4 BUFADR,
2            ITMFLAGS
  END MAP
  MAP                            ! A longword of 0 terminates
   INTEGER*4 END_LIST            !  an item list
  END MAP
 END UNION
END STRUCTURE
RECORD /PSCANITMLST/             ! Declare the item list for
2         PSCANLIST(5)           !  $PROCESS_SCAN

RECORD /JPIITMLST/               ! Declare the item list for
2         JPILIST(6)             !  $GETJPI

INTEGER*4 SYS$GETJPIW,           ! System service entry points
2         SYS$PROCESS_SCAN

INTEGER*4 STATUS,                ! Status variable
2         CONTEXT,               ! Context from $PROCESS_SCAN
2         PID,                   ! PID from $GETJPI
2         JPIFLAGS               ! Flags for $GETJPI

INTEGER*2 IOSB(4)                ! I/O Status Block for $GETJPI

CHARACTER*16
2         PRCNAM,                ! Process name from $GETJPI
2         NODENAME               ! Node name from $GETJPI
INTEGER*2 PRCNAM_LEN,            ! Process name length
2         NODENAME_LEN           ! Node name length

CHARACTER*80
2         IMAGNAME               ! Image name from $GETJPI
INTEGER*2 IMAGNAME_LEN           ! Image name length

LOGICAL*4 DONE                   ! Done with data loop
```

(continued on next page)

Example 3–12 (Cont.) Using SYS$GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
!********************************************
!*  Initialize item list for $PROCESS_SCAN  *
!********************************************

! Look for interactive and batch jobs across
! the cluster with $GETJPI buffering

PSCANLIST(1).BUFLEN   = 0
PSCANLIST(1).CODE     = PSCAN$_NODE_CSID
PSCANLIST(1).BUFADR   = 0
PSCANLIST(1).ITMFLAGS = PSCAN$M_NEQ
PSCANLIST(2).BUFLEN   = 0
PSCANLIST(2).CODE     = PSCAN$_MODE
PSCANLIST(2).BUFADR   = JPI$K_INTERACTIVE
PSCANLIST(2).ITMFLAGS = PSCAN$M_OR
PSCANLIST(3).BUFLEN   = 0
PSCANLIST(3).CODE     = PSCAN$_MODE
PSCANLIST(3).BUFADR   = JPI$K_BATCH
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).BUFLEN   = 0
PSCANLIST(4).CODE     = PSCAN$_GETJPI_BUFFER_SIZE
PSCANLIST(4).BUFADR   = 1000
PSCANLIST(4).ITMFLAGS = 0
PSCANLIST(5).END_LIST = 0

!********************************************
!*     End of item list initialization      *
!********************************************

STATUS = SYS$PROCESS_SCAN (        ! Set up the scan context
2                         CONTEXT,
2                         PSCANLIST)

IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

! Initialize $GETJPI item list

JPILIST(1).BUFLEN    = 4
JPILIST(1).CODE      = IAND ('FFFF'X, JPI$_GETJPI_CONTROL_FLAGS)
JPILIST(1).BUFADR    = %LOC(JPIFLAGS)
JPILIST(1).RETLENADR = 0
JPILIST(2).BUFLEN    = 4
JPILIST(2).CODE      = JPI$_PID
JPILIST(2).BUFADR    = %LOC(PID)
JPILIST(2).RETLENADR = 0
JPILIST(3).BUFLEN    = LEN(PRCNAM)
JPILIST(3).CODE      = JPI$_PRCNAM
JPILIST(3).BUFADR    = %LOC(PRCNAM)
JPILIST(3).RETLENADR = %LOC(PRCNAM_LEN)
JPILIST(4).BUFLEN    = LEN(IMAGNAME)
JPILIST(4).CODE      = JPI$_IMAGNAME
JPILIST(4).BUFADR    = %LOC(IMAGNAME)
JPILIST(4).RETLENADR = %LOC(IMAGNAME_LEN)
JPILIST(5).BUFLEN    = LEN(NODENAME)
JPILIST(5).CODE      = JPI$_NODENAME
JPILIST(5).BUFADR    = %LOC(NODENAME)
JPILIST(5).RETLENADR = %LOC(NODENAME_LEN)
JPILIST(6).END_LIST  = 0
! Loop calling $GETJPI with the context

DONE = .FALSE.
JPIFLAGS = IOR (JPI$M_NO_TARGET_INSWAP, JPI$M_IGNORE_TARGET_STATUS)
DO WHILE (.NOT. DONE)
```

(continued on next page)

Example 3–12 (Cont.) Using SYS$GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
                ! Call $GETJPI to get the next process

                STATUS = SYS$GETJPIW (
        2                    ,            ! No event flag
        2                    CONTEXT,     ! Process context
        2                    ,            ! No process name
        2                    JPILIST,     ! Itemlist
        2                    IOSB,        ! Always use IOSB with $GETJPI!
        2                    ,            ! No AST
        2                    )            ! No AST arg
                ! Check the status in both STATUS and the IOSB, if
                ! STATUS is OK then copy IOSB(1) to STATUS

                IF (STATUS) STATUS = IOSB(1)

                ! If $GETJPI worked, display the process, if done then
                ! prepare to exit, otherwise signal an error

                IF (STATUS) THEN
                        IF (IMAGNAME_LEN .EQ. 0) THEN
                                TYPE 1010, PID, NODENAME, PRCNAM
                        ELSE
                                TYPE 1020, PID, NODENAME, PRCNAM,
        2                                    IMAGNAME(1:IMAGNAME_LEN)
                        END IF
                ELSE IF (STATUS .EQ. SS$_NOMOREPROC) THEN
                        DONE = .TRUE.
                ELSE
                        CALL LIB$SIGNAL(%VAL(STATUS))
                END IF

        END DO

1010    FORMAT (' ',Z8.8,'  ',A6,':: ',A,' (no image)')
1020    FORMAT (' ',Z8.8,'  ',A6,':: ',A,' ',A)

        END
```

3.2.7 Using SYS$GETLKI

The SYS$GETLKI system service allows you to obtain process lock information. Example 3–13 is a C program that illustrates the procedure for obtaining process lock information for both Alpha and VAX systems. However, to compile on Alpha systems, you need to supply the /DEFINE=Alpha=1 qualifier.

Example 3–13 Procedure for Obtaining Process Lock Information

```
#pragma nostandard
#ifdef  Alpha
#pragma module               LOCK_SCAN
#else             /* Alpha */
#module                      LOCK_SCAN
#endif           /* Alpha */
#pragma standard

#include        <ssdef.h>
#include        <lkidef.h>
```

(continued on next page)

Example 3–13 (Cont.) Procedure for Obtaining Process Lock Information

```
#pragma nostandard
globalvalue
      ss$_normal, ss$_nomorelock;
#pragma standard

struct lock_item_list
                {
                short int       buffer_length;
                short int       item_code;
                void            *bufaddress;
                void            *retaddress;
                };

typedef struct lock_item_list lock_item_list_type;

unsigned long lock_id;
long int value_block[4];

#pragma nostandard
static  lock_item_list_type
  getlki_item_list[] = {
   {sizeof(value_block), LKI$_VALBLK,    &value_block,  0},
   {sizeof(lock_id),     LKI$_LOCKID,    &lock_id,      0},
   {0,0,0,0}
};
globalvalue ss$_normal, ss$_nomorelock;
#pragma standard

main()
{
  int status = ss$_normal;
  unsigned long lock_context = -1;   /* init for wild-card operation */

  while (status == ss$_normal) {
      status = sys$getlkiw( 1, &lock_context, getlki_item_list,0,0,0,0);
      /*                                                    */
      /* Dequeue the lock if the value block contains a 1 */
      /*                                                    */
      if ((status == ss$_normal) & (value_block[0] == 1)){
          status = sys$deq( lock_id, 0, 0, 0 );
      }
  }
  if (status != ss$_nomorelock){
      exit(status);
  }
}
```

3.2.8 Setting Process Privileges

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

3.3 Changing Process and Kernel Threads Scheduling

Prior to kernel threads, the OpenVMS scheduler selected a process to run. With kernel threads, the OpenVMS scheduler selects a kernel thread to run. All processes are thread capable processes with at least one kernel thread. A process may have only one kernel thread, or a process may have a variable number of kernel threads. A single-threaded process is equivalent to a process before OpenVMS V7.0.

With kernel threads, all base and current priorities are per-kernel thread. To alter a thread's scheduling, you can change the base priority of the thread with the SYS$SETPRI system service, which affects the specified kernel thread and not the entire process.

To alter a process's scheduling, you can lock the process into physical memory so that it is not swapped out. Processes that have been locked into physical memory are executed before processes that have been swapped out. For kernel threads, the thread with the highest priority level is executed first.

If you create a subprocess with the LIB$SPAWN routine, you can set the priority of the subprocess by executing the DCL command SET PROCESS/PRIORITY as the first command in a command procedure. You must have the ALTPRI privilege to increase the base priority above the base priority of the creating process.

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

If you alter kernel thread's scheduling, you must do so with care. Review the following considerations before you attempt to alter the standard kernel threads or process scheduling with either SYS$SETPRI or SYS$SETSWM:

- Priority—Increasing a kernel threads's base priority gives that thread more processor time at the expense of threads that execute at lower priorities. Digital does not recommended this unless you have a program that must respond immediately to events (for example, a real-time program). If you must increase your base priority, return it to normal as soon as possible. If the entire image must execute at an increased priority, reset the base priority before exiting; image termination does not reset the base priority.
- Swapping—Inhibiting swapping keeps your process in physical memory. Digital does not recommended this 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).

3.4 Using Affinity and Capabilities in CPU Scheduling

Alpha

The **affinity** and **capabilities** mechanism allows CPU scheduling to be adapted to larger CPU configurations by controlling the distribution of processes, or threads, throughout the active CPU set. The control of the distribution of processes throughout the active CPU set becomes more important as higher-performance server applications, such as databases and real-time, process-control environments, are implemented. Affinity and capabilities provide the user with the opportunities to perform the following:

- Create and modify a set of user-defined process capabilities

- Create and modify a set of user-defined CPU capabilities to match those in the process
- Allow a process to apply the affinity mechanisms to a subset of the active CPU set in a symmetric multiprocessing (SMP) configuration

3.4.1 Defining Affinity and Capabilities

The affinity mechanism allows a process, or each of its kernel threads, to specify an exact set of CPUs on which it can execute. The capabilities mechanism allows a process to specify a set of resources that a CPU in the active set must have defined before it is allowed to contend for the process execution. Today, both of these mechanisms are present in the OpenVMS scheduling mechanism; both are used extensively internally and externally to implement parts of the I/O and timing subsystems. Now, however, the OpenVMS operating system provides user access to these mechanisms.

3.4.1.1 Using Affinity and Capabilities with Caution

It is important for the user to understand that inappropriate and abusive use of the affinity and capabilities mechanisms can have a negative impact on the symmetric aspects of the current multi-CPU scheduling algorithm.

3.4.2 Types of Capabilities

Capabilities are resources assigned to CPUs that a process needs to execute correctly. There are four defined capabilities. They are restricted to internal system events or functions that control system states or functions. Table 3–6 describes the four capabilities.

Table 3–6 Capabilities

Capability	Description
Primary	This capability is owned by only one CPU at a time, since the primary could possibly migrate from CPU to CPU in the configuration. The system requires for I/O and timekeeping functions that the process run on the primary CPU. The process requiring this capability is only allowed to run on the processor that has it at the time.
Run	This capability controls the ability of a CPU to execute any process at all. Every process requires this resource and if the CPU does not have it, scheduling for that CPU comes to a halt in a recognized state. The command STOP/CPU uses this capability when it is trying to make the CPU quiescent, bringing it to a halted state.
Quorum	This capability is used in a clustered environment when a node wants another node to come to a quiescent state until further notice. Like the Run capability, Quorum is a required resource for every process and every CPU for scheduling to occur.
Vector	This capability is obsolete on OpenVMS Alpha systems, but is retained as a compatibility feature with OpenVMS VAX. Like the Primary capability, it reflects a feature of the CPU; that is, that the CPU has a vector processing unit directly associated with it.

3.4.3 Looking at User Capabilities

Previously, the use of capabilities was restricted solely to system resources and control events. It is valuable, however, for user functions to have the ability to indicate a resource or special CPU function as well as the system.

There are 16 user-defined capabilities added to both the process and CPU structures. Unlike the static definitions of the current system capabilities, the user capabilities have meaning only in the context of the processes that define them. Through system service interfaces, processes or individual threads of a multi-thread process, have the ability to set specific bits in the capability masks of a CPU to give it a resource, and to set specific bits in the kernel thread's capability mask to require that resource as an execution criteria.

The user capability feature is a direct superset of the current capability functionality. All currently existing capabilities are placed into the system capability set; they are not available to the process through system service interfaces. These system service interfaces affect only the 16 bits specifically set aside for user definition.

The OpenVMS operating system has no direct knowledge of what the defined capability is being used. All responsibility for the correct definition, use and management of these bits is determined by the processes that define them. The system controls the impact of these capabilities through privilege requirements; but, as with the priority adjustment services, abusive use of the capability bits could affect the scheduling dynamic and CPU loads in an SMP environment.

3.4.4 Using the Capabilities System Services

The SYS$CPU_CAPABILITIES and SYS$PROCESS_CAPABILITIES system services provide access to the capability features. By using the SYS$CPU_ CAPABILITIES and SYS$PROCESS_CAPABILITIES services, you can assign user capabilities to a CPU and specific kernel thread. Assigning a user capability to a CPU lasts for the life of the system, or until another explicit change is made. This operation has no direct effect on the scheduling dynamics of the system; it only indicates that the specified CPU is capable of handling any process, or thread, that requires that particular resource. If a process does not indicate it needs that resource, it ignores the CPU's additional capability and schedules the process based on other process requirements.

Assigning a user capability requirement to a specific process, or thread, has a major impact on the scheduling state of that entity. For the process, or thread, to be scheduled on a CPU in the active set, that CPU must have the capability assigned prior to the scheduling attempt. If no CPU currently has the correct set of capability requirements, the process is placed into a wait state until a CPU becomes available with the right configuration. Like system capabilities, user process capabilities are additive; that is, for a CPU to schedule the process it must have the full complement of required capabilities.

These services reference both sets of 16-bit user capabilities by the common symbolic constant names of CAP$M_USER1 through CAP$M_USER16. These names reflect the corresponding bit position in the appropriate capability mask; they are non-zero and self-relative to themselves only.

Both services allow multiple bits to be set or cleared, or both, simultaneously. Each take as parameters a select and modify mask that define the operation set to be performed. The service callers are responsible for setting up the select mask to indicate which of the user capabilities bits are to be affected by the current call. This select mask is a bit vector of the ORed bit symbolic names which, when set, states the value in the modify mask is the new value of the bit. Both masks use the symbolic constants to indicate the same bit, or, if appropriate, the symbolic constant CAP$K_USER_ALL can be used in the select mask to indicate that the entire set of capabilities is affected. Likewise, the symbolic constant CAP$K_USER_ADD or CAP$K_USER_REMOVE can be used in the modify mask to indicate all capabilities specified in the select mask are to be set or cleared respectively.

For information on using the SYS$CPU_CAPABILITIES and SYS$PROCESS_CAPABILITIES, see the *OpenVMS System Services Reference Manual: A–GETMSG* and *OpenVMS System Services Reference Manual: GETQUI–Z*.

3.4.5 Types of Affinity

The two types of affinity are implicit and explicit affinity. This section presents a description of each.

3.4.5.1 Implicit Affinity

Implicit affinity, sometimes known as soft affinity, is a variant form of the original affinity mechanism used in the OpenVMS scheduling mechanisms. Rather than require a process to stay on a specific CPU regardless of conditions, implicit affinity maximizes cache and translation buffer (TB) context by maintaining an association with the CPU that has the most information about a given process.

Currently, the OpenVMS scheduling mechanism already has a version of implicit affinity. It keeps track of the last CPU the process ran on and tries to schedule itself to that CPU, subject to a fairness algorithm. The fairness algorithm makes sure a process is not skipped too many times when it normally would have been scheduled elsewhere.

The Alpha architecture lends itself to maintaining cache and TB context that has significant potential for performance improvement with both the process and system level. Because this feature contradicts the normal highest priority process scheduling algorithms in an SMP configuration, implicit affinity cannot be a system default.

The system service, SYS$SET_IMPLICIT_AFFININTY, provides implicit affinity support. This service works on an explicitly specified process or kernel thread block (KTB) through the **pidadr** and **prcnam** arguments. The default is the current process, but if the symbolic constant CAP$K_PROCESS_DEFAULT is specified in the **pidadr**, the bit is set in the global default cell SCH$GL_DEFAULT_PROCESS_CAP. Setting implicit affinity globally is similar to setting a capability bit in the same mask, because every process creation after the modification picks up the bit as a default that stays in effect across all image activations.

The protections required to invoke SYS$SET_IMPLICIT_AFFINITY depend on the process that is being affected. Because the addition of implicit affinity has the same potential for affecting the priority scheduling of processes in the COM queue as that of the SYS$ALTPRI service, ALTPRI protection is required as the base which all modification forms of the serve must have to invoke SYS$SET_IMPLICIT_AFFINITY. If the process is the current one, no other privilege is required. To affect processes in the same UIC group, the GROUP privilege is required. For any other processes in the system, the WORLD privilege is required.

3.4.5.2 Explicit Affinity

Even though capabilities and affinity have much overlap in their functional behavior, they are, nonetheless, two discrete scheduling mechanisms. Affinity, the subsetting of the number of CPUs on which a process can execute, has precedence over the capability feature and provides an explicit binding operation between the process and CPU. It forces the scheduling algorithm to consider only the CPU set it requires, and then applies the capability tests to see if any of them are appropriate.

Explicit affinity allows database and high-performance applications to segregate application functions to individual CPUs, providing improved cache and TB performance, as well as reducing context switching and general scheduling overhead. During the IPL 8 scheduling pass, the process is investigated to see to which CPUs it is bound and whether the current CPU is one of those. If it passes that test, capabilities are also validated to allow the process to context switch. The number of CPUs that can be supported is 32.

The SYS$PROCESS_AFFINITY system service provides access to the explicit affinity functionality. SYS$PROCESS_AFFINITY resolves to a specific process, defaulting to the current one, through the **pidadr** and **prcnam** arguments. Like the other system services, the CPUs that are affected are indicated through select_mask, and the binding state of each CPU is specified in modify_mask.

Specific CPUs can be referenced in the select_mask and modify_mask using the symbolic constants CAP$M_CPU0 through CAP$M_CPU31. These constants are defined to match the bit position of their associated CPU ID. Alternatively, specifying CAP$K_ALL_ACTIVE_CPUS in select_mask sets or clears explicit affinity for all CPUs in the current active set.

Explicit affinity, like capabilities, has a permanent process as well as current image copy. As each completed image is run down, the permanent explicit affinity values overwrite the running image set, superseding any changes that were made in the interim. Specifying CAP$M_FLAG_PERMANENT in the flags parameter indicates that both the current and permanent processes are to be modified simultaneously. As a result, unless explicitly changed again, this operation has a scope from the current image through the end of the process life.

For information on using the SYS$SET_IMPLICIT_AFFINITY and SYS$PROCESS_AFFINITY, see the *OpenVMS System Services Reference Manual: A–GETMSG* and *OpenVMS System Services Reference Manual: GETQUI–Z*. ♦

3.5 Changing Process Name

Use the system service SYS$SETPRN to change the name of your process. SYS$SETPRN can be used only on the calling process. Changing process names might be useful when a lengthy image is being executed. You can change names at significant points in the program; then monitor program execution through the change in process names. You can obtain a process name by calling a SYS$GETJPI routine from within a controlling process, by pressing the Ctrl/T key sequence if the image is currently executing in your process, or by entering the DCL command SHOW SYSTEM if the program is executing in a detached process.

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

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

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

3.6 Accessing Another Process's Context

VAX

On OpenVMS VAX systems, a system programmer must sometimes develop code that performs various actions (such as performance monitoring) on behalf of a given process, executing in that process's context. To do so, a programmer typically creates a routine consisting of position-independent code and data, allocates sufficient space in nonpaged pool, and copies the routine to it. On OpenVMS VAX systems, such a routine can execute correctly no matter where it is loaded into memory. ♦

Alpha

On OpenVMS Alpha systems, the practice of moving code in memory is more difficult and complex. It is not enough to simply copy code from one memory location to another. On OpenVMS Alpha systems, you must relocate both the routine and its linkage section, being careful to maintain the relative distance between them, and then apply all appropriate fixups to the linkage section.

The OpenVMS Alpha system provides two mechanisms to enable one process to access the context of another:

- Code that must read from or write to another process's registers or address space can use the system routines EXE$READ_PROCESS and EXE$WRITE_ PROCESS as described in Section 3.6.1.
- Code that must perform other operations in another process's context (for instance, to execute a system service to raise a target process's quotas) can be written as an OpenVMS Alpha executive image as described in Section 3.6.2. ♦

3.6.1 Reading and Writing in the Address Space of Another Process (Alpha Only)

Alpha

EXE$READ_PROCESS and EXE$WRITE_PROCESS are OpenVMS Alpha system routines in nonpaged system space. EXE$READ_PROCESS reads data from a target process's address space or registers and writes it to a buffer in the local process's address space. EXE$WRITE_PROCESS obtains data from a local process's address space and transfers it to the target process's context. Both routines must be called from kernel mode at IPL 0.

One of the arguments to these procedures specifies whether or not the procedure is to access memory and registers in the target process. Another argument specifies the memory address or register number. The contents of these arguments are symbolic names (beginning with the prefix EACB$) that are defined by the $PROCESS_READ_WRITE macro in SYS$LIBRARY:LIB.MLB. (They are also defined in LIB.REQ for BLISS programmers.)

3.6.1.1 EXE$READ_PROCESS and EXE$WRITE_PROCESS

The following are descriptions of the callable interfaces to EXE$READ_PROCESS and EXE$WRITE_PROCESS.

EXE$READ_PROCESS

Reads data from a target process's address space or registers and writes it to a buffer in the local process's address space.

Module

PROC_READ_WRITE

Format

status = EXE$READ_PROCESS (ipid, buffer_size, target_address, local_address, target_address_type, ast_counter_address)

Arguments

ipid

OpenVMS usage	ipid
type	longword (unsigned)
access	read only
mechanism	by value

Internal process ID of the target process.

buffer_size

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by value

Number of bytes to transfer. If **target_address_type** is EACB$K_GENERAL_REGISTER, the values of **target_address** and **buffer_size** together determine how many 64-bit registers are written, in numeric order, to the buffer. A partial register is written for any value that is not a multiple of 8.

If you specify **buffer_size** to be larger than 8, more than one register is written from the buffer. Registers are written in numeric order, followed by the PC and PS, starting at the register indicated by **target_address**.

If **target_address_type** is EACB$K_GENERAL_REGISTER and the values of **buffer_size** and **target_address** would cause a target process read extending beyond R31 (the last available register), EXE$READ_PROCESS returns SS$_ILLSER status.

target_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference (if address) by value (if constant)

If **target_address_type** is EACB$K_MEMORY, address in target process at which the transfer is to start.

If **target_address_type** is EACB$K_GENERAL_REGISTER, symbolic constant indicating at which general register the transfer should start. Possible constant values include EACB$K_R0 through EACB$K_R29, EACB$K_PC, and EACB$K_PS.

local_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference

Address of buffer in local process to which data is to be written.

target_address_type

OpenVMS usage	integer
type	longword (unsigned)
access	read only
mechanism	by value

Symbolic constant indicating whether the **target_address** argument is a memory address (EACB$K_MEMORY) or a general register (EACB$K_GENERAL_REGISTER). Floating-point registers are not supported as target addresses.

ast_counter_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference

Address of a longword used internally as an AST counter by EXE$READ_PROCESS and EXE$WRITE_PROCESS to detect errors. Supply the same address in the **ast_counter_address** argument for every call to these routines.

Returns

OpenVMS usage	cond_value
type	longword (unsigned)
access	write only
mechanism	by value

Status indicating the success or failure of the operation.

Return Values

SS$_ACCVIO	Unable to write to the location indicated by **local_address** or **ast_counter_address**.
SS$_ILLSER	Routine was called with IPL greater than 0, or an illegal **target_address_type** was specified. If **target_address_type** is EACB$K_GENERAL_REGISTER, this status can indicate that the values of **buffer_size** and **target_address** would cause a target process read extending beyond R31 (the last available register).
SS$_INSFMEM	Insufficient memory available for specified buffer.
SS$_NONEXPR	The **ipid** argument does not correspond to an existing process.
SS$_NORMAL	The interprocess read finished successfully.

SS$_TIMEOUT	The read operation did not finish within a few seconds.

Context

The caller of EXE$READ_PROCESS must be executing in kernel mode at IPL 0. Kernel mode ASTs must be enabled.

Description

EXE$READ_PROCESS reads data from a target process's address space and writes it to a buffer in the local process's address space.

EXE$READ_PROCESS allocates nonpaged pool for an AST control block (ACB), an ACB extension, and a buffer of the specified size. It initializes the extended ACB with information describing the data transfer and then delivers an AST to the target process to perform the operation. The data is read in the context of the target process from its address space or registers into nonpaged pool. An AST is then queued to the requesting process to complete the read operation by copying the data from pool to the process's buffer.

EXE$READ_PROCESS does not return to its caller until the read is completed, an error is encountered, or it has timed out. (The current timeout value is 3 seconds.)

EXE$WRITE_PROCESS

Reads data from the local process's address space and writes it to a target process's registers or a buffer in a target process's address space.

Module

PROC_READ_WRITE

Format

status = EXE$WRITE_PROCESS (ipid, buffer_size, local_address, target_address,
target_address_type, ast_counter_address)

Arguments

ipid

OpenVMS usage	ipid
type	longword (unsigned)
access	read only
mechanism	by value

Internal process ID of the target process.

buffer_size

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by value

Number of bytes to transfer. If **target_address_type** is EACB$K_GENERAL_REGISTER, the values of **target_address** and **buffer_size** together determine how many 64-bit registers are written, in numeric order, from the buffer. A partial register is written for any value that is not a multiple of 8.

If you specify **buffer_size** to be larger than 8, more than one register is written from the buffer. Registers are written in numeric order, followed by the PC and PS, starting at the register indicated by **target_address**.

If **target_address_type** is EACB$K_GENERAL_REGISTER and the values of **buffer_size** and **target_address** would cause a write extending beyond R31 (the last available register), EXE$WRITE_PROCESS returns SS$_ILLSER status.

local_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference

Address in local process from which data is to be transferred.

target_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference (if address) by value (if constant)

If **target_address_type** is EACB$K_MEMORY, address in target process at which the transfer is to start.

If **target_address_type** is EACB$K_GENERAL_REGISTER, symbolic constant indicating at which general register the transfer should start. Possible constant values include EACB$K_R0 through EACB$K_R29, EACB$K_PC, and EACB$K_PS.

target_address_type

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by value

Symbolic constant indicating whether the **target_address** argument is a memory address (EACB$K_MEMORY) or a general register (EACB$K_GENERAL_REGISTER). Floating-point registers are not supported as target addresses.

ast_counter_address

OpenVMS usage	longword_unsigned
type	longword (unsigned)
access	read only
mechanism	by reference

Address of a longword used internally as an AST counter by EXE$READ_PROCESS and EXE$WRITE_PROCESS to detect errors. Supply the same address in the **ast_counter_address** argument for every call to these routines.

Returns

OpenVMS usage	cond_value
type	longword (unsigned)
access	write only
mechanism	by value

Status indicating the success or failure of the operation.

Return Values

SS$_ACCVIO	Unable to read from the location indicated by **local_address** or write to the location indicated by **ast_counter_address**.
SS$_ILLSER	Routine was called with IPL greater than 0, an illegal **target_address_type** was specified. If **target_address_type** is EACB$K_GENERAL_REGISTER, this status can indicate that the values of **buffer_size** and **target_address** would cause a process write extending beyond R31 (the last available register).
SS$_INSFMEM	Insufficient memory available for specified buffer.
SS$_NONEXPR	The **ipid** argument does not correspond to an existing process.
SS$_NORMAL	The interprocess write finished successfully.
SS$_TIMEOUT	The write operation did not finish within a few seconds.

Context

The caller of EXE$WRITE_PROCESS must be executing in kernel mode at IPL 0. Kernel mode ASTs must be enabled.

Description

EXE$WRITE_PROCESS reads data from the local process's address space and writes it to a target process's registers or a buffer in a target process's address space.

EXE$WRITE_PROCESS allocates nonpaged pool for an AST control block (ACB), an ACB extension, and a buffer of the specified size. It initializes the extended ACB with information describing the data transfer, copies the data to be written to the target process into the buffer, and then delivers an AST to the target process to perform the operation.

The AST routine copies the data from pool into the target location and then queues an AST to the requesting process to complete the write operation.

EXE$WRITE_PROCESS does not return to its caller until the read is completed, an error is encountered, or it has timed out. (The current timeout value is 3 seconds.)

♦

3.6.2 Writing an Executive Image (Alpha Only)

Alpha

An **executive image** is an image that is mapped into system space as part of the OpenVMS executive. It contains data, routines, and initialization code specific to an image's functions and features. An executive image is implemented as a form of shareable image. Like any shareable image, it has a global symbol table, image section descriptors, and an image activator fixup section. Unlike a shareable image, however, an executive image does not contain a symbol vector.

Universally available procedures and data cells in system-supplied executive images are accessed through entries provided by the symbol vectors in the system base images SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE. References to an executive image are resolved through these symbol vectors, whether from an executive image or from a user executable or shareable image.

Unlike a system-supplied executive image, an executive image that you create cannot provide universally accessible entry points and symbols in this manner. Instead, it must establish its own vector of procedure descriptors for its callable routines and make the address of that vector available systemwide.

The OpenVMS executive loader imposes several requirements on the sections of any executive image. These requirements include the following:

- An executive image can contain at most one image section of the following types and no others:
  - Nonpaged execute section (for code)
  - Nonpaged read/write section (for read-only and writable data, locations containing addresses that must be relocated at image activation, and the linkage section for nonpaged code)
  - Paged execute section (for code)
  - Paged read/write section (for read-only and writable data, locations containing addresses that must be relocated at image activation, and the linkage section for pageable code)
  - Initialization section (for initialization procedures and their associated linkage section and data)
  - Image activator fixup section

  The modules of an executive image define program sections (PSECTS) with distinct names. The named PSECTS are necessary so that the program sections can be collected into clusters, by means of the COLLECT= linker option, during linking. A COLLECT= option specified in the linking of an executive image generates each of the first five image sections.

  The image activator fixup section is created by the linker to enable the image activator to finally resolve references to SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE with addresses within the loaded executive image. The OpenVMS executive loader deallocates the memory for both the fixup section and the initialization section once the executive image has been initialized.

- In your linker options file, you use the COLLECT= option to collect the program sections from the modules comprising the executive image and place them into one of the standard clusters of an executive image, as described above, marked with the appropriate attributes. If the attributes of a PSECT being collected into an image section do not match the attributes of that section, you use the PSECT_ATTR= option to force the attributes to match.

All PSECTS must have the same attributes so that only one image section is produced from the cluster.

Note that, in OpenVMS Alpha systems, the execute section cannot contain data. You must collect all data, whether read-only or writable, into one of the read/write sections.

- You link an executive image as a type of shareable image that can be loaded into system space by the executive loader. You may link an executive image in such a way (using the /SECTION_BINDING qualifier to the LINK command) that the executive loader consolidates its image sections into granularity hint regions within memory with similar image sections of the executive. This process yields a tangible performance benefit on OpenVMS Alpha systems. See the *OpenVMS Linker Utility Manual* for more information on section binding.

  See Section 3.6.2.2 for a template of a LINK command and linker options file used to produce an executive image.

An executive image can contain one or more initialization procedures that are executed when the image is loaded. If the image is listed in SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DAT as created by means of System Management utility (SYSMAN) commands, initialization procedures can be run at various stages of system initialization.

An initialization routine performs a variety of functions, some specific to the features supported by the image and others required by many executive images. An executive image declares an initialization routine by invoking the INITIALIZATION_ROUTINE macro (see Section 3.6.2.1).

Because an initialization routine is executed when the executive image is loaded, it serves as an ideal mechanism for locating the callable routines of an executive image in memory. One method for accomplishing this involves the following procedure:

- Establishing a vector table within the executive image. For instance, the MACRO-32 Compiler for OpenVMS Alpha would compile the following into the address of a vector table consisting of the procedure values of routines residing elsewhere in the image:

```
ROUTINE_VECTOR:
.ADDRESS        ROUTINE_A       ;address of routine A
.ADDRESS        ROUTINE_B       ;address of routine B
   .
   .
   .
.ADDRESS        ROUTINE_M       ;address of routine M
```

- Declaring an initialization procedure using the INITIALIZATION_ROUTINE macro.
- Loading the image by calling LDR$LOAD_IMAGE as described in Section 3.6.2.4, with LDR$V_USER_BUF set in the flags longword that is the second argument to LDR$LOAD_IMAGE and with the address of a user buffer in the third argument. The dynamic loader passes the address of the user buffer to the initialization procedure.

  Alternatively, you can use the System Management utility (SYSMAN) to load the executive image. Using a SYSMAN command to update the OpenVMS system images file (SYS$LOADABLE_IMAGES:VMS$SYSTEM_

IMAGES.DATA) allows the image to be loaded and initialized at various phases of system initialization (see Section 3.6.2.3).

- Inserting code in the initialization routine that locates the vector table and stores it in a place accessible to users of routines in the executive image.

  An initialization routine can accomplish this by returning the address of the vector table to the caller of LDR$LOAD_IMAGE in the specified user buffer. The caller of LDR$LOAD_IMAGE can take whatever steps necessary to make the routines accessible to potential callers. One such mechanism is to call the Create Logical Name ($CRELNM) system service to establish a systemwide logical name whose equivalence name is the address of the vector table.

For additional information about OpenVMS executive images, see *VMS for Alpha Platforms Internals and Data Structures*. ♦

3.6.2.1 INITIALIZATION_ROUTINE Macro (Alpha Only)

Alpha

The following describes the invocation format of the INITIALIZATION_ROUTINE macro. An equivalent macro, $INITIALIZATION_ROUTINE is provided for BLISS programmers. For C programmers, INIT_RTN_SETUP.H in SYS$LIB_C.TLB is available.

INITIALIZATION_ROUTINE

Declares a routine to be an initialization routine.

Format

INITIALIZATION_ROUTINE name [,system_rtn=0] [,unload=0] [,priority=0]

Parameters

name
Name of the initialization routine.

[system_rtn=0]
Indicates whether the initialization routine is external to the module invoking the macro. The default value of 0, indicating that the initialization routine is part of the current module, is the only option supported on OpenVMS Alpha systems.

[unload=0]
Indicates whether the **name** argument specifies an unload routine. The default value of 0, indicating that the argument specifies an initialization routine, is the only option supported on OpenVMS Alpha systems.

[priority=0]
Indicates the PSECT in which the entry for this initialization routine should be placed. Routines that specify the **priority** argument as 0 are placed in the first PSECT (EXEC$INIT_000); those that specify a value of 1 are placed in the second (EXEC$INIT_001). The routines in the first PSECT are called before those in the second.

Description

The INITIALIZATION_ROUTINE macro declares a routine to be an initialization routine. It enters a vector for the routine in the initialization routine vector table, guaranteeing that this routine is called when the image is loaded by the OpenVMS loader.

♦

3.6.2.2 Linking an Executive Image (Alpha Only)

Alpha

The following template can serve as the basis of a LINK command and linker options file used to create an OpenVMS executive image. See the *OpenVMS Linker Utility Manual* for a full description of most linker qualifiers and options referenced by this example.

Note

Use of the linker to create executive images (specifically the use of the /ATTRIBUTES switch on the COLLECT= option) is not documented elsewhere and is not supported by Digital.

```
! Replace 'execlet' with your image name

$ LINK/ALPHA/USERLIB=LNK$LIBRARY/NATIVE_ONLY/BPAGES=14 -
 /REPLACE/SECTION/NOTRACEBACK-
 /SHARE=EXE$:execlet-
 /MAP=MAP$:execlet /FULL /CROSS -
 /SYMBOL=EXE$:execlet -
 SYS$INPUT/OPTION, -
!
SYMBOL_TABLE=GLOBALS
! Creates .STB for System Dump Analyzer
CLUSTER=execlet,,,- 1
ALPHA$LIBRARY:STARLET/INCLUDE:(SYS$DOINIT) - 2
! Insert executive object code here
ALPHA$LOADABLE_IMAGES:SYS$BASE_IMAGE.EXE/SHAREABLE/SELECTIVE
PSECT_ATTR=EXEC$INIT_LINKAGE,PIC,USR,CON,REL,GBL,NOSHR,EXE,RD,WRT,NOVEC  3
!
COLLECT=NONPAGED_READONLY_PSECTS/ATTRIBUTES=RESIDENT,-  4
 nonpaged_code
COLLECT=NONPAGED_READWRITE_PSECTS/ATTRIBUTES=RESIDENT,-
 nonpaged_data,-
 nonpaged_linkage
COLLECT=PAGED_READONLY_PSECTS,-
 paged_code,-
 $CODE$
COLLECT=PAGED_READWRITE_PSECTS,-
 paged_data,-
 paged_linkage,-
 $LINK$,-
 $PLIT$,-
 $INITIAL$,-
 $LITERAL$,-
 $GLOBAL$,-
 $OWN$,-
 $DATA$
COLLECT=INITIALIZATION_PSECTS/ATTRIBUTES=INITIALIZATION_CODE,-
 EXEC$INIT_CODE,-
 EXEC$INIT_LINKAGE,-
 EXEC$INIT_000,-
 EXEC$INIT_001,-
 EXEC$INIT_002,-
 EXEC$INIT_SSTBL_000,-
 EXEC$INIT_SSTBL_001,-
 EXEC$INIT_SSTBL_002
```

1 The CLUSTER= option creates the named cluster *execlet*, specifying the order in which the linker processes the listed modules.

2 The object module SYS$DOINIT (in STARLET.OLB) is explicitly linked into an executive image. This module declares the initialization routine table and provides routines to drive the actual initialization of an executive image.

3 The image sections in an executive image are generated according to the attributes assigned to program sections within source modules by the compilers and linker. Because an executive image can consist only of a certain number and type of image section, the options file contains PSECT_ATTR= options to coerce the attributes of various program sections contributing to the image so that the linker produces only one image section from the cluster. Correspondingly, the options file consists of a set of COLLECT= options clauses to collect these program sections into each permitted image section.

4 This series of COLLECT= options is required to create and assign image section attributes (RESIDENT or INITIALIZATION_CODE) to the NONPAGED_READONLY_PSECTS, NONPAGED_READWRITE_PSECTS, and INITIALIZATION_PSECTS image sections of an executive image. Each COLLECT= option collects a set of program sections with similar attributes into a cluster that the linker uses to produce one of the permitted image sections for an executive image. ♦

3.6.2.3 Loading an Executive Image (Alpha Only)

Alpha

There are two methods of loading an executive image:

- Calling LDR$LOAD_IMAGE to load the executive image at run time. This method lets you pass the address of a buffer to the image's initialization routine by which the caller and the initialization routine can exchange data. Section 3.6.2.4 describes LDR$LOAD_IMAGE. Note that you must link the code that calls LDR$LOAD_IMAGE against the system base image, using the /SYSEXE qualifier to the LINK command.
- Using the SYSMAN SYS_LOADABLE ADD command and updating the OpenVMS system images file (SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA). This method causes the executive image to be loaded and initialized during the phases of system initialization. (See *VMS for Alpha Platforms Internals and Data Structures* for information about how an executive image's initialization routine is invoked during system initialization.)

To load an executive image with the System Management utility (SYSMAN), perform the following tasks:

1. Copy the executive image to SYS$LOADABLE_IMAGES.
2. Add an entry for the executive image in the data file SYS$UPDATE:VMS$SYSTEM_IMAGES.IDX, as follows:

   ```
   SYSMAN SYS_LOADABLE ADD _LOCAL_ executive-image -
   /LOAD_STEP = SYSINIT -
   /SEVERITY  = WARNING -
   /MESSAGE   = "failure to load executive-image"
   ```

3. Invoke the SYS$UPDATE:VMS$SYSTEM_IMAGES.COM command procedure to generate a new system image data file (file name SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA). During the bootstrap, the system uses this data file to load the appropriate images.
4. Reboot the system. This causes the new executive image to be loaded into the system. ♦

3.6.2.4 LDR$LOAD_IMAGE (Alpha Only)

Alpha

The following is a description of the callable interface to LDR$LOAD_IMAGE.

LDR$LOAD_IMAGE

Loads an OpenVMS executive image into the system.

Module

SYSLDR_DYN

Format

LDR$LOAD_IMAGE filename ,flags ,ref_handle ,user_buf

Arguments

filename

OpenVMS usage	character string
type	character string
access	read only
mechanism	by descriptor

The longword address of a character string descriptor containing the file name of the executive image to be loaded. The file name can include a directory specification and image name, but no device name. If you omit the directory specification, LDR$LOAD_IMAGE supplies SYS$LOADABLE_IMAGES as the default.

flags

OpenVMS usage	flags
type	longword (unsigned)
access	read only
mechanism	value

A flags longword, containing the following bit fields. (Symbolic names for these bit fields are defined by the $LDRDEF macro in SYS$LIBRARY:LIB.MLB.)

Bit Field	Description
LDR$V_PAG	When set, indicates that the image should be loaded with its pageable sections resident. The flag is generally based on the value of the bit 0 in the S0_PAGING system parameter.
LDR$V_UNL	When set, indicates that the image may be removed from memory.
LDR$V_OVR	When set, indicates that the image's read-only sections should not be overwritten during bugcheck processing. This flag is unused on OpenVMS Alpha systems.
LDR$V_USER_BUF	When set, indicates that the caller has passed the address of a buffer that should be passed to the initialization routine.

Bit Field	Description
LDR$V_NO_SLICE	When set, indicates that the image's sections should not be loaded into a granularity hint region.

ref_handle

OpenVMS usage	address
type	longword (signed)
access	write only
mechanism	by reference

The longword address of a reference handle, a three-longword buffer to be filled by LDR$LOAD_IMAGE as follows:

+00	Address of loaded image or zero if image was loaded sliced.
+04	Address of loaded image data block (LDRIMG). See the $LDRIMGDEF macro definition in SYS$LIBRARY:LIB.MLB and *VMS for Alpha Platforms Internals and Data Structures* for a description of the LDRIMG structure.
+08	Loaded image sequence number.

user_buf

OpenVMS usage	address
type	longword (signed)
access	read only
mechanism	by reference

The longword address of a user buffer passed to executive image's initialization routine if LDR$V_USER_BUF is set in the flags longword.

Context

LDR$LOAD_IMAGE must be called in executive mode.

Returns

OpenVMS usage	cond_value
type	longword (unsigned)
access	write only
mechanism	by value

Status indicating the success or failure of the operation.

Return Values

SS$_ACCVIO	Unable to read the character string descriptor containing the file name of the executive image to be loaded or to write the reference handle.
LOADER$_BAD_GSD	An executive image was loaded containing a global symbol that is not vectored through either the SYS$BASE_IMAGE or the SYS$PUBLIC_ VECTORS image.

SS$_BADIMGHDR	Image header is larger than two blocks or was built with a version of the linker that is incompatible with LDR$LOAD_IMAGE.
LOADER$_BADIMGOFF	During a sliced image load request, a relocation or fixup operation was attempted on an image offset that has no resultant address within the image.
LOADER$_DZRO_ISD	A load request was made for an executive image that illegally contains demand zero sections.
SS$_INSFARG	Fewer than three arguments were passed to LDR$LOAD_IMAGE, or, with LDR$V_USER_BUF set in the flags longword, fewer than four arguments.
SS$_INSFMEM	Insufficient nonpaged pool for the LDRIMG structure or insufficient physical memory to load nonpageable portion of an executive image (that is, an image loaded as nonsliced).
SS$_INSFSPTS	Insufficient system page table entries (SPTEs) to describe the address space required for the executive image to be loaded as nonsliced.
LOADER$_MULTIPLE_ISDS	A load request was made for an image that was not linked correctly because it contains more than one each of the following types of sections: fixup initialization nopaged code nopaged data paged code paged data
LOADER$_NO_PAGED_ ISDS	SYSBOOT failed to load the executive image because it contains either paged code or paged data sections.
SS$_NOPRIV	LDR$LOAD_IMAGE was called from user or supervisor mode.
LOADER$_NO_SUCH_ IMAGE	A load request was made for an executive image that was linked against a shareable image that is not loaded. The only legal shareable images for executive images are SYS$BASE_IMAGE and SYS$PUBLIC_VECTORS.
SS$_NORMAL	Normal, successful completion.
LOADER$_PAGED_GST_ TOBIG	An executive image has more global symbols in the fixup data than can fit in the loader's internal tables.
LOADER$_PSB_FIXUPS	A load request was made for an executive image that contains LPPSB fixup because it was linked /NONATIVE_ONLY. Executive images must be linked /NATIVE_ONLY.

LOADER$_SPF_TOBIG	The loader's internal tables cannot accommodate all of the executive image fixups that must be postponed to later in the bootstrap operation.
SS$_SYSVERDIF	Image was linked with versions of executive categories incompatible with those of the running system.
LOADER$_VEC_TOBIG	An attempt to load an executive image failed because the image's symbol vector updates for SYS$BASE_IMAGE and SYS$PUBLC_ VECTORS exceed the size of the loader's internal tables.
Other	Any RMS error status returned from $OPEN failures. Any I/O error status returned from $QIO failures.

Description

LDR$LOAD_IMAGE loads an executive image into system space and calls its initialization routine. Optionally, LDR$LOAD_IMAGE returns to its caller information about the loaded image.

The initialization routine is passed two or three longword arguments, depending upon the setting of LDR$V_USER_BUF:

- Address of loaded image data block (LDRIMG)
- The flags longword
- The longword address of a user buffer passed to the executive image's initialization routine (if LDR$V_USER_BUF is set in the flags longword)

♦

3.6.2.5 LDR$UNLOAD_IMAGE (Alpha Only)

Alpha The following is a description of the callable interface to LDR$UNLOAD_IMAGE.

LDR$UNLOAD_IMAGE

Unloads a removable executive image. This routine is called to unload an execlet. All resources are returned.

Module

SYSLDR_DYN

Format

LDR$UNLOAD_IMAGE   filename ,ref_handle

Arguments

filename

OpenVMS usage	character string
type	character string
access	read only
mechanism	by descriptor

The longword address of a character string descriptor containing the file name of the executive image to be unloaded. The file name must be supplied exactly as it was supplied to LDR$LOAD_IMAGE when the executive image was loaded.

ref_handle

OpenVMS usage	address
type	longword (signed)
access	read only
mechanism	by reference

The longword address of the reference handle containing the three-longword block returned by LDR$LOAD_IMAGE when the executive image was loaded.

Context

LDR$UNLOAD_IMAGE must be called in kernel mode.

Returns

OpenVMS usage	cond_value
type	longword (unsigned)
access	write only
mechanism	by value

Status indicating the success or failure of the operation.

Return Values

SS$_INSFARG	LDR$UNLOAD_IMAGE was not called with two parameters.
SS$_BADPARAMS	Reference handle data inconsistent with LDRIMG block that matches the name in the first argument.
LOADER$_MARKUNL	A call was made to the LDR$UNLOAD_IMAGE routine to unload a removable executive image that already has an outstanding unload request against it.
SS$_NOPRIV	LDR$UNLOAD_IMAGE was not called in kernel mode.
SS$_NORMAL	Executive image was successfully removed from the system.
LOADER$_NOT_UNL	A call was made to LDR$UNLOAD_IMAGE to unload an executive image that is not loaded or that was not loaded with the LDR$V_UNL flag bit set.

LOADER$_UNL_PEN	A call was made to LDR$UNLOAD_IMAGE to unload an executive image that is in use. The image is marked to be unloaded later.

Description

LDR$UNLOAD_IMAGE removes an executive image from system space and returns all memory resources allocated when the image was loaded. Images can only be removed if they were originally loaded with the bit LDR$V_UNL set in the input flags to LDR$LOAD_IMAGE.

♦

3.7 Synchronizing Programs by Specifying a Time for Program Execution

You can synchronize timed program execution in the following ways:

- Executing a program at a specific time
- Executing a program at timed intervals

3.7.1 Obtaining the System Time

The process control procedures that allow you to synchronize timed program execution require you to supply a system time value.

You can use either system services or RTL routines for obtaining and reading time. They are summarized in Table 3–7. With these routines, you can determine the system time, convert it to an external time, and pass a time back to the system. The system services use the operating system's default date format. With the RTL routines, you can use the default format or specify your own date format. However, if you are just using the time and date for program synchronization, using the operating system's default format is probably sufficient.

When using the RTL routines to change date/time formats, initialization routines are required. Refer to the *OpenVMS RTL Library (LIB$) Manual* for more information.

See Chapter 5 for a further discussion of using timing operations with the operating system.

Table 3–7 Time Manipulation System Services and Routines

Routine	Description
SYS$GETTIM	Obtains the current date and time in 64-bit binary format
SYS$NUMTIM	Converts system date and time to numeric integer values
SYS$ASCTIM	Converts an absolute or delta time from 64-bit system time format to an ASCII string
SYS$ASCUTC	Converts an absolute time from 128-bit Coordinated Universal Time (UTC) format to an ASCII string
LIB$SYS_ASCTIM	Converts binary time to ASCII string
SYS$BINTIM	Converts a date and time from ASCII to system format
SYS$BINUTC	Converts an ASCII string to an absolute time value in the 128-bit UTC format
SYS$FAO	Converts a binary value into an ASCII character string in decimal, hexadecimal, or octal notation and returns the character string in an output string
SYS$GETUTC	Returns the current time in 128-bit UTC format
SYS$NUMUTC	Converts an absolute 128-bit binary time into its numeric components. The numeric components are returned in local time
SYS$TIMCON	Converts 128-bit UTC to 64-bit system format or 64-bit system format to 128-bit UTC based on the value of the convert flag

(continued on next page)

Table 3–7 (Cont.) Time Manipulation System Services and Routines

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

3.7.1.1 Executing a Program at a 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. Because you do not want the parent process to remain in hibernation until the process executes, execute the process concurrently.

You can also use the SYS$CREPRC system service to execute a program at a specified time. However, because a process created by SYS$CREPRC hibernates rather than terminates after executing the desired program, Digital recommends you use the LIB$SPAWN routine unless you need a detached process.

Example 3–14 executes a program at a specified delta time. The parent program prompts the user for a delta time, equates the delta time to the symbol EXECUTE_TIME, and then creates a subprocess to execute the command procedure LATER.COM. LATER.COM uses the symbol EXECUTE_TIME as the parameter for the WAIT command. (You might also allow the user to enter an absolute time and have your program change it to a delta time by subtracting the current time from the specified time. Chapter 5 discusses time manipulation.)

Example 3–14 Executing a Program Using Delta Time

```
! Delta time
CHARACTER*17 TIME
INTEGER LEN
```

(continued on next page)

Example 3–14 (Cont.) Executing a Program Using Delta Time

```
! 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
```

3.7.1.2 Executing a Program at Timed Intervals

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

Using LIB$SPAWN

Using LIB$SPAWN, you can create a subprocess that executes a command procedure containing three commands: the DCL command WAIT, the command that invokes the desired program, and a GOTO command that directs control back to the WAIT command. Because you do not want the parent process to remain in hibernation until the subprocess executes, execute the subprocess concurrently. See Section 3.7.1.1 for an example of LIB$SPAWN.

Using SYS$CREPRC

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

1. Create and hibernate a process—Use SYS$CREPRC to create a process that executes the desired program. Set the PRC$V_HIBER bit of the **stsflg** argument of the SYS$CREPRC system service to indicate that the created process should hibernate before executing the program.
2. Schedule a wakeup call for the created subprocess—Use the SYS$SCHDWK system service to specify the time at which the system should wake up the subprocess, and a time interval at which the system should repeat the wakeup call.

Example 3–15 executes a program at timed intervals. The program creates a subprocess that immediately hibernates. (The identification number of the created subprocess is returned to the parent process so that it can be passed to SYS$SCHDWK.) The system wakes up the subprocess at 6:00 a.m. on the 23rd (month and year default to system month and year) and every 10 minutes thereafter.

Example 3–15 Executing a Program at Timed Intervals

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

3.7.2 Placing Entries in the System Timer Queue

When you use the system timer queue, you use the timer expiration to signal when a routine is to be executed. It allows the caller to request a timer that will activate some time in the future. The timer is requested for the calling kernel thread. When the timer activates, the event is reported to that thread. It does not affect any other thread in the process.

For the actual signal, you can use an event flag or AST. With this method, you do not need a separate process to control program execution. However, you do use up your process's quotas for ASTs and timer queue requests.

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

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

Specifying the Starting Time

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

Signaling Timer Expiration

Once the system has reached this time, the timer expires. To signal timer expiration, set an event flag in the **efn** argument or specify an AST routine to be executed in the **astadr** argument. Refer to Section 14.6 and Chapter 4 for more information about using event flags and ASTs.

How Timer Requests Are Identified

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

To cancel a timer request, use the SYS$CANTIM system service.

3.8 Suspending, Resuming, and Stopping Kernel Threads and Process Execution

You can control kernel threads and process execution in the following ways:

- Suspending a process—All kernel threads associated with the specified process are suspended.
- Hibernating a process—Only the calling kernel thread is hibernated.
- Stopping a process—All kernel threads associated with the specified process are stopped.
- Resuming a process—All kernel threads associated with the specified process are resumed.
- Exiting an image—All kernel threads associated with the specified process are exited.
- Deleting a process—All kernel threads associated with the specified process are deleted, and then the process is deleted.

3.8.1 Process Hibernation and Suspension

There are two ways to halt the execution of a kernel thread or process temporarily:

- Hibernation—Performed by the Hibernate (SYS$HIBER) system service, which affects the calling kernel thread.
- Suspension—Performed by the Suspend Process (SYS$SUSPND) system service, which affects all of the kernel threads associated with the specified process.

The kernel thread can continue execution normally only after a corresponding Wake from Hibernation (SYS$WAKE) system service (if it is hibernating), or after a Resume Process (SYS$RESUME) system service, if it is suspended.

Suspending or hibernating a kernel thread puts it into a dormant state; the thread is not deleted.

A process in hibernation can control itself; a process in suspension requires another process to control it. Table 3–8 compares hibernating and suspended processes.

Table 3–8 Process Hibernation and Suspension

Hibernation	Suspension
Can only cause self to hibernate	Can suspend self or another process, depending on privilege; it suspends all threads associated with the specified process
Reversed by SYS$WAKE /SYS$SCHDWK system service	Reversed by SYS$RESUME system service
Interruptible; can receive ASTs	Noninterruptible; cannot receive ASTs[1]
Can wake self	Cannot cause self to resume
Can schedule wakeup at an absolute time or at a fixed time interval	Cannot schedule resumption

[1]If a process is suspended at kernel mode (a hard suspension), it cannot receive any ASTs. If a process is suspended at supervisor mode (a soft suspension), it can receive executive or kernel mode ASTs. See the description of SYS$SUSPND in the *OpenVMS System Services Reference Manual: GETQUI–Z*.

Table 3–9 summarizes the system services and routines that can place a process in or remove from hibernation or suspension.

Table 3–9 System Services and Routines Used for Hibernation and Suspension

Routine	Function
Hibernating Processes	
SYS$HIBER	Places the requesting kernel thread in the hibernation state. An AST can be delivered to the thread while it is hibernating. The service only puts the calling thread into HIB; no other thread is affected.
SYS$WAKE	Resumes execution of a kernel thread in hibernation. This service wakes all hibernating kernel threads in a process regardless of the caller. Any thread that is not hibernating when the service is called is marked *wake pending*. Because of the *wake pending*, the next call to SYS$HIBER completes immediately and the thread does not hibernate. Premature wakeups must be handled in the code.
SYS$SCHDWK	Resumes execution of a kernel thread in hibernation at a specified time. This service schedules a wakeup request for a thread that is about to call SYS$HIBER. The wakeup only affects the requesting thread; any other hibernating kernel threads are not affected.
LIB$WAIT	Uses the services SYS$SCHDWK and SYS$HIBER.
SYS$CANWAK	Cancels a scheduled wakeup issued by SYS$SCHDWK. Unless called with a specific timer request ID, this service cancels all timers for all threads in the process regardless of the calling thread.

(continued on next page)

Table 3–9 (Cont.) System Services and Routines Used for Hibernation and Suspension

Routine	Function
Suspended Kernel Threads and Processes	
SYS$SUSPEND	Puts in a suspended state all threads associated with the specified process
SYS$RESUME	Puts in an execution state all threads of the specified process

3.8.1.1 Using Process Hibernation

The hibernate/wake mechanism provides an efficient way to prepare an image for execution and then to place it in a wait state until it is needed.

If you create a subprocess that must execute the same function repeatedly and must execute immediately when it is needed, you could use the SYS$HIBER and SYS$WAKE system services, as shown in the following example:

```
/* Process TAURUS */

#include <stdio.h>
#include <descrip.h>

main() {

        unsigned int status;
        $DESCRIPTOR(prcnam,"ORION");
        $DESCRIPTOR(image,"COMPUTE.EXE");

/* Create ORION */
        status = SYS$CREPRC(0,                1         /* Process id */
                            &image,                       /* Image */
                            0, 0, 0, 0, 0,
                            &prcnam,                      /* Process name */
                            0, 0, 0, 0);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
/* Wake ORION */
        status = SYS$WAKE(0, &prcnam);          2
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
/* Wake ORION again */
        status = SYS$WAKE(0, &prcnam);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
}

/* Process ORION and image COMPUTE */
```

```
#include <stdio.h>
#include <ssdef.h>
   .
   .
   .
sleep:
        status = SYS$HIBER();                 3
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        goto sleep;
}
```

1 Process TAURUS creates the process ORION, specifying the descriptor for the image named COMPUTE.

2 At an appropriate time, TAURUS issues a SYS$WAKE request for ORION. ORION continues execution following the SYS$HIBER service call. When it finishes its job, ORION loops back to repeat the SYS$HIBER call and to wait for another wakeup.

3 The image COMPUTE is initialized, and ORION issues the SYS$HIBER system service.

The Schedule Wakeup (SYS$SCHDWK) system service, a variation of the SYS$WAKE system service, schedules a wakeup for a hibernating process at a fixed time or at an elapsed (delta) time interval. Using the SYS$SCHDWK service, a process can schedule a wakeup for itself before issuing a SYS$HIBER call. For an example of how to use the SYS$SCHDWK system service, see Chapter 5.

Hibernating processes can be interrupted by asynchronous system traps (ASTs), as long as AST delivery is enabled. The process can call SYS$WAKE on its own behalf in the AST service routine, and continue execution following the execution of the AST service routine. For a description of ASTs and how to use them, see Chapter 4.

3.8.1.2 Using Alternative Methods of Hibernation

You can use two additional methods to cause a process to hibernate:

- Specify the **stsflg** argument for the SYS$CREPRC system service, setting the bit that requests SYS$CREPRC to place the created process in a state of hibernation as soon as it is initialized.
- Specify the /DELAY, /SCHEDULE, or /INTERVAL qualifier to the RUN command when you execute the image from the command stream.

When you use the SYS$CREPRC system service, the creating process can control when to wake the created process. When you use the RUN command, its qualifiers control when to wake the process.

If you use the /INTERVAL qualifier and the image to be executed does not call the SYS$HIBER system service, the image is placed in a state of hibernation whenever it issues a return instruction (RET). Each time the image is awakened, it begins executing at its entry point. If the image does call SYS$HIBER, each time it is awakened it begins executing at either the point following the call to SYS$HIBER or at its entry point (if it last issued a RET instruction).

If wakeup requests are scheduled at time intervals, the image can be terminated with the Delete Process (SYS$DELPRC) or Force Exit (SYS$FORCEX) system service, or from the command level with the STOP command. The SYS$DELPRC and SYS$FORCEX system services are described in Section 3.8.3.4 and in Section 3.8.4. The RUN and STOP commands are described in the *OpenVMS DCL Dictionary*.

These methods allow you to write programs that can be executed once, on request, or cyclically. If an image is executed more than once in this manner, normal image activation and termination services are not performed on the second and subsequent calls to the image. Note that the program must ensure both the integrity of data areas that are modified during its execution and the status of opened files.

3.8.1.3 Using SYS$SUSPND

Using the Suspend Process (SYS$SUSPND) system service, a process can place itself or another process into a wait state similar to hibernation. Suspension, however, is a more pronounced state of hibernation. The operating system provides no system service to force a process to be swapped out, but the SYS$SUSPND system service can accomplish the task in the following way. Suspended processes are the first processes to be selected for swapping. A suspended process cannot be interrupted by ASTs, and it can resume execution only after another process calls a Resume Process (SYS$RESUME) system service on its behalf. If ASTs are queued for the process while it is suspended, they are delivered when the process resumes execution. This is an effective tool for blocking delivery of all ASTs.

At the DCL level, you can suspend a process by issuing the SET PROCESS command with the /SUSPEND qualifier. This command temporarily stops the process's activities. The process remains suspended until another process resumes or deletes it. To allow a suspended process to resume operation, use either the /NOSUSPEND or /RESUME qualifier.

3.8.2 Passing Control to Another Image

The RTL routines LIB$DO_COMMAND and LIB$RUN_PROGRAM allow you to invoke the next image from the current image. That is, they allow you to perform image rundown 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.

3.8.2.1 Invoking a Command Image

The following DCL 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 (RTL) routine LIB$DO_COMMAND. If LIB$DO_COMMAND executes successfully, control is not returned to the invoking image, and statements following the LIB$DO_COMMAND statement are not executed. The following statement causes the current image to exit and executes the DCL command in the preceding example:

```
   .
   .
   .
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
```

3.8.2.2 Invoking a Noncommand Image

You invoke a noncommand image 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
```

3.8.3 Performing Image Exit

When image execution completes normally, the operating system performs a variety of image rundown functions. If the image is executed by the command interpreter, image rundown prepares the process for the execution of another image. If the image is not executed by the command interpreter—for example, if it is executed by a subprocess—the process is deleted.

Main programs and main routines terminate by executing a return instruction (RET). This instruction returns control to the caller, which could have been LIB$INITIALIZE, the debugger, or the command interpreter. The completion code, SS$_NORMAL, which has the value 1, should be used to indicate normal successful completion.

Any other condition value can be used to indicate success or failure. The command language interpreter uses the condition value as the parameter to the Exit (SYS$EXIT) system service. If the severity field (STS$V_SEVERITY) is SEVERE or ERROR, the continuation of a batch job or command procedure is affected.

These exit activities are also initiated when an image completes abnormally as a result of any of the following conditions:

- Specific error conditions caused by improper specifications when a process is created. For example, if an invalid device name is specified for the SYS$INPUT, SYS$OUTPUT, or SYS$ERROR logical name, or if an invalid or nonexistent image name is specified, the error condition is signaled in the created process.
- An exception occurring during execution of the image. When an exception occurs, any user-specified condition handlers receive control to handle the exception. If there are no user-specified condition handlers, a system-declared condition handler receives control, and it initiates exit activities for the image. Condition handling is described in Chapter 13.
- A Force Exit (SYS$FORCEX) system service issued on behalf of the process by another process.

3.8.3.1 Performing Image Rundown

The operating system performs image rundown functions that release system resources obtained by a process while it is executing in user mode. These activities occur in the following order:

1. Any outstanding I/O requests on the I/O channels are canceled, and I/O channels are deassigned.
2. Memory pages occupied or allocated by the image are deleted, and the working set size limit of the process is readjusted to its default value.
3. All devices allocated to the process at user mode are deallocated (devices allocated from the command stream in supervisor mode are not deallocated).
4. Timer-scheduled requests, including wakeup requests, are canceled.
5. Common event flag clusters are disassociated.
6. Locks are dequeued as a part of rundown.
7. User mode ASTs that are queued but have not been delivered are deleted, and ASTs are enabled for user mode.
8. Exception vectors declared in user mode, compatibility mode handlers, and change mode to user handlers are reset.
9. System service failure exception mode is disabled.
10. All process private logical names and logical name tables created for user mode are deleted. Deletion of a logical name table causes all names in that table to be deleted. Note that names entered in shareable logical name tables, such as the job or group table, are not deleted at image rundown, regardless of the access mode for which they were created.

3.8.3.2 Initiating Rundown

To initiate the rundown activities described in Section 3.8.3.1, the system calls the Exit (SYS$EXIT) system service on behalf of the process. In some cases, a process can call SYS$EXIT to terminate the image itself (for example, if an unrecoverable error occurs).

You should not call the SYS$EXIT system service directly from a main program. By not calling SYS$EXIT directly from a main program, you allow the main program to be more like ordinary modular routines and therefore usable by other programmers as callable routines.

The SYS$EXIT system service accepts a status code as an argument. If you use SYS$EXIT to terminate image execution, you can use this status code argument to pass information about the completion of the image. If an image returns without calling SYS$EXIT, the current value in R0 is passed as the status code when the system calls SYS$EXIT.

This status code is used as follows:

- The command interpreter uses the status code to display optionally an error message when it receives control following image rundown.
- If the image has declared an exit handler, the status code is written in the address specified in the exit control block.
- If the process was created by another process, and the creator has specified a mailbox to receive a termination message, the status code is written into the termination mailbox when the process is deleted.

3.8.3.3 Performing Cleanup and Rundown Operations

Use exit handlers to perform image-specific cleanup or rundown operations. For example, if an image uses memory to buffer data, an exit handler can ensure that the data is not lost when the image exits as the result of an error condition.

To establish an exit-handling routine, you must set up an exit control block and specify the address of the control block in the call to the Declare Exit Handler (SYS$DCLEXH) system service. You can call an exit handler by using standard calling conventions; you can provide arguments to the exit handler in the exit control block. The first argument in the control block argument list must specify the address of a longword for the system to write the status code from SYS$EXIT.

If an image declares more than one exit handler, the control blocks are linked together on a last-in, first-out (LIFO) basis. After an exit handler is called and returns control, the control block is removed from the list. You can remove exit control blocks prior to image exit by using the Cancel Exit Handler (SYS$CANEXH) system service.

Exit handlers can be declared from system routines executing in supervisor or executive mode. These exit handlers are also linked together in other lists, and they receive control after exit handlers that are declared from user mode are executed.

Exit handlers are called as a part of the SYS$EXIT system service. While a call to the SYS$EXIT system service often precedes image rundown activities, the call is not a part of image rundown. There is no way to ensure that exit handlers will be called if an image terminates in a nonstandard way.

3.8.3.4 Initiating Image Rundown for Another Process

The Force Exit (SYS$FORCEX) system service provides a way for a process to initiate image rundown for another process. For example, the following call to SYS$FORCEX causes the image executing in the process CYGNUS to exit:

```
        $DESCRIPTOR(prcnam,"CYGNUS");
   .
   .
   .
        status = SYS$FORCEX(0,                   /* pidadr - Process id */
                            &prcnam,         /* prcnam - Process name */
                    0);                      /* code - Completion code */
```

Because the SYS$FORCEX system service calls the SYS$EXIT system service, any exit handlers declared for the image are executed before image rundown. Thus, if the process is using the command interpreter, the process is not deleted and can run another image. Because the SYS$FORCEX system service uses the AST mechanism, an exit cannot be performed if the process being forced to exit has disabled the delivery of ASTs. AST delivery and how it is disabled and reenabled is described in Chapter 4.

The following program segment shows an example of an exit-handling routine:

```
#include <stdio>
#include <ssdef>

/* Exit control block */

struct {
        unsigned int *desblk;
        unsigned int (*exh)();
        unsigned int argcount;
        unsigned int *cond_value;             1
}exitblock = {0, &exitrtn, 1, 0};

main() {

        unsigned int status;

/* Declare the exit handler */

        status = SYS$DCLEXH(&exitblock);        2
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
}

int exitrtn (int condition ) {                  3
        if ((status & 1) != 1)
        {
        /* Clean up */
   .
   .
   .
                return 1;
        }
        else
        /* Normal exit */

                return 0;

}
```

1 EXITBLOCK is the exit control block for the exit handler EXITRTN. The third longword indicates the number of arguments to be passed. In this example, only one argument is passed: the address of a longword for the system to store the return status code. This argument must be provided in an exit control block.

2 The SYS$DCLEXH system service call designates the address of the exit control block, thus declaring EXITRTN as an exit handler.

3 The EXITRTN exit handler checks the status code. If this is a normal exit, EXITRTN returns control. Otherwise, it handles the error condition.

3.8.4 Deleting a Process

Process deletion completely removes a process from the system. A process can be deleted by any of the following events:

- The Delete Process (SYS$DELPRC) system service is called.
- A process that created a subprocess is deleted.
- An interactive process uses the DCL command LOGOUT.
- A batch job reaches the end of its command file.
- An interactive process uses the DCL command STOP/ID=*pid* or STOP *username*.
- A process that contains a single image calls the Exit (SYS$EXIT) system service.
- The Force Exit (SYS$FORCEX) system service forces image exit on a process that contains a single image.

When the system is called to delete a process as a result of any of these conditions, it first locates all subprocesses, and searches hierarchically. No process can be deleted until all the subprocesses it has created have been deleted.

The lowest subprocess in the hierarchy is a subprocess that has no descendant subprocesses of its own. When that subprocess is deleted, its parent subprocess becomes a subprocess that has no descendant subprocesses and it can be deleted as well. The topmost process in the hierarchy becomes the parent process of all the other subprocesses.

The system performs each of the following procedures, beginning with the lowest process in the hierarchy and ending with the topmost process:

- The image executing in the process is run down. The image rundown that occurs during process deletion is the same as that described in Section 3.8.3.1. When a process is deleted, however, the rundown releases all system resources, including those acquired from access modes other than user mode.
- Resource quotas are released to the creating process, if the process being deleted is a subprocess.
- If the creating process specifies a termination mailbox, a message indicating that the process is being deleted is sent to the mailbox. For detached processes created by the system, the termination message is sent to the system job controller.
- The control region of the process's virtual address space is deleted. (The control region consists of memory allocated and used by the system on behalf of the process.)
- All system-maintained information about the process is deleted.

Figure 3–1 illustrates the flow of events from image exit through process deletion.

Figure 3–1 Image Exit and Process Deletion

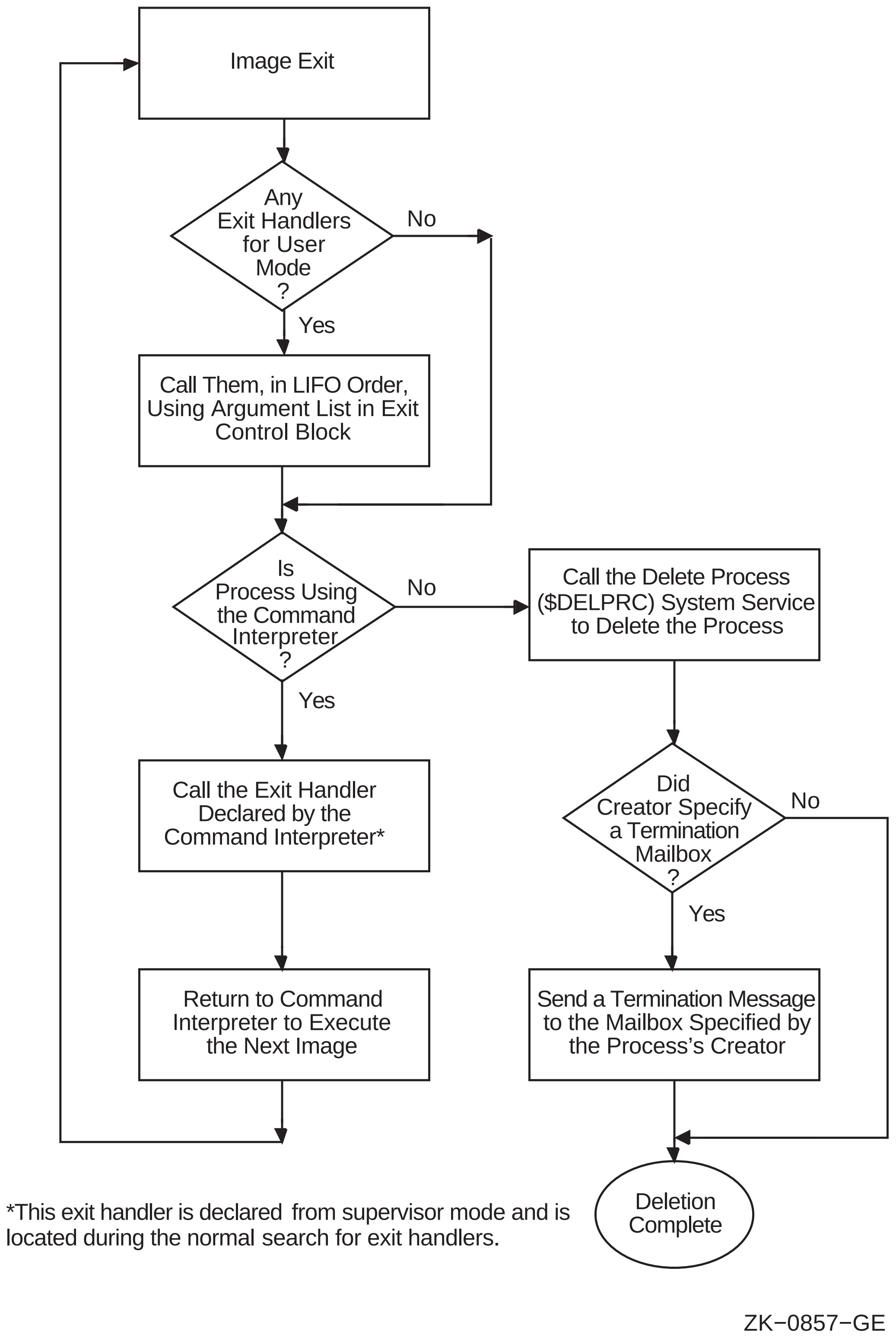


Image Exit
Any Exit Handlers for User Mode ?
No
Yes
Call Them, in LIFO Order, Using Argument List in Exit Control Block
Is Process Using the Command Interpreter ?
No
Call the Delete Process ($DELPRC) System Service to Delete the Process
Yes
Call the Exit Handler Declared by the Command Interpreter*
Did Creator Specify a Termination Mailbox ?
No
Yes
Return to Command Interpreter to Execute the Next Image
Send a Termination Message to the Mailbox Specified by the Process's Creator
Deletion Complete
*This exit handler is declared from supervisor mode and is located during the normal search for exit handlers.
ZK−0857−GE

]

3.8.4.1 Deleting a Process By Using System Services

A process can delete itself or another process at any time, depending on the restrictions outlined in Section 3.1.1. Any one of the following system services can be used to delete a subprocess or a detached process. Some services terminate execution of the image in the process; others terminate the process itself.

- SYS$EXIT—Initiates normal exit in the current image. Control returns to the command language interpreter. If there is no command language interpreter, the process is terminated. This routine cannot be used to terminate an image in a detached process.
- SYS$FORCEX—Initiates a normal exit on the image in the specified process. GROUP or WORLD privilege may be required, depending on the process specified. An AST is sent to the specified process. The AST calls on the SYS$EXIT routine to complete the image exit. Because an AST is used, you cannot use this routine on a suspended process. You can use this routine on a subprocess or detached process. See Section 3.8.3.4 for an example.
- SYS$DELPRC—Deletes the specified process. GROUP or WORLD privilege may be required, depending on the process specified. A termination message is sent to the calling process's mailbox. You can use this routine on a subprocess, a detached process, or the current process. For example, if a process has created a subprocess named CYGNUS, it can delete CYGNUS, as follows:

```
$DESCRIPTOR(prcnam,"CYGNUS");
   .
   .
   .
status = SYS$EDLPRC(0,                 /* Process id */
                    &prcnam);          /* Process name */
```

  Because a subprocess is automatically deleted when the image it is executing terminates (or when the command stream for the command interpreter reaches end of file), you normally do not need to call the SYS$DELPRC system service explicitly.

3.8.4.2 Terminating Mailboxes

A termination mailbox provides a process with a way of determining when, and under what conditions, a process that it has created was deleted. The Create Process (SYS$CREPRC) system service accepts the unit number of a mailbox as an argument. When the process is deleted, the mailbox receives a termination message.

The first word of the termination message contains the symbolic constant, MSG$_DELPROC, which indicates that it is a termination message. The second longword of the termination message contains the final status value of the image. The remainder of the message contains system accounting information used by the job controller and is identical to the first part of the accounting record sent to the system accounting log file. The description of the SYS$CREPRC system service in the *OpenVMS System Services Reference Manual* provides the complete format of the termination message.

If necessary, the creating process can determine the process identification of the process being deleted from the I/O status block (IOSB) posted when the message is received in the mailbox. The second longword of the IOSB contains the process identification of the process being deleted.

A termination mailbox cannot be located in memory shared by multiple processors.

The following example illustrates a complete sequence of process creation, with a termination mailbox:

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <msgdef.h>
#include <dvidef.h>
#include <iodef.h>
#include <accdef.h>

unsigned short unitnum;
unsigned int pidadr;

/* Create a buffer to store termination info */

struct accdef exitmsg;

/* Define and initialize the item list for $GETDVI */

static struct {                                               1
        unsigned short buflen,item_code;
        void *bufaddr;
        void *retlenaddr;
        unsigned int terminator;
}mbxinfo = { 4, DVI$_UNIT, &unitnum, 0, 0};

/* I/O Status Block for QIO */

struct {
        unsigned short iostat, mblen;
        unsigned int mbpid;
}mbxiosb;

main() {

        void exitast(void);
        unsigned short exchan;
        unsigned int status,maxmsg=84,bufquo=240,promsk=0;
        unsigned int func=IO$_READVBLK;
        $DESCRIPTOR(image,"LYRA");

/* Create a mailbox */
        status = SYS$CREMBX(0,      /* prmflg (permanent or temporary) */ 2
                      &exchan,      /* channel */
                      maxmsg,       /* maximum message size */
                      bufquo,       /* no. of bytes used for buffer */
                      promsk,       /* protection mask */
                      0,0,0,0);
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );

/* Get the mailbox unit number */
        status = SYS$GETDVI(0,                  /* efn - event flag */      3
                      exchan,           /* chan - channel */
                      0,                /* devnam - device name */
                      &mbxinfo,         /* item list */
                      0,0,0,0);
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );
/* Create a subprocess */
        status = SYS$CREPRC(&pidadr,    /* process id */
                        &image,         /* image to be run */
                        0,0,0,0,0,0,0,0,
                        unitnum,        /* mailbox unit number */
                0);                     /* options flags */
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );

/* Read from mailbox */
        status = SYS$QIOW(0,            /* efn - event flag */           4
                        exchan,         /* chan - channel number */
                        func,           /* function modifier */
                        &mbxiosb,       /* iosb - I/O status block */
                        &exitast,       /* astadr - astadr AST routine */
                0,                      /* astprm - astprm AST parameter */
                        &exitmsg,       /* p1 - buffer to receive message*/
                        ACC$K_TERMLEN,  /* p2 - length of buffer */
                0,0,0,0);               /* p3, p4, p5, p6 */

        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );

}
void exitast(void) {
     if(mbxiosb.iostat == SS$_NORMAL)                        5
     {
         printf("\nMailbox successfully written...");
         if (exitmsg.acc$w_msgtyp == MSG$_DELPROC)
         {
              printf("\nProcess deleted...");
              if (pidadr == mbxiosb.mbpid)
              {
                    printf("\nPIDs are equal...");
                    if (exitmsg.acc$l_finalsts == SS$_NORMAL)
                      printf("\nNormal termination...");
                    else
                     printf("\nAbnormal termination status: %d",
                            exitmsg.acc$l_finalsts);
               }
               else
                     printf("\nPIDs are not equal");
               }
               else
                    printf("\nTermination message not received... status: %d",
                           exitmsg.acc$w_msgtyp);
        }
        else
                printf("\nMailbox I/O status block: %d",mbxiosb.iostat);

        return;
}
```

1 The item list for the Get Device/Volume Information (SYS$GETDVI) system service specifies that the unit number of the mailbox is to be returned.

2 The Create Mailbox and Assign Channel (SYS$CREMBX) system service creates the mailbox and returns the channel number at EXCHAN.

3 The Create Process (SYS$CREPRC) system service creates a process to execute the image LYRA.EXE and returns the process identification at LYRAPID. The **mbxunt** argument refers to the unit number of the mailbox, obtained from the Get Device/Volume Information (SYS$GETDVI) system service.

4 The Queue I/O Request (SYS$QIO) system service queues a read request to the mailbox, specifying both an AST service routine to receive control when the mailbox receives a message and the address of a buffer to receive the message. The information in the message can be accessed by the symbolic offsets defined in the $ACCDEF macro. The process continues executing.

5 When the mailbox receives a message, the AST service routine EXITAST receives control. Because this mailbox can be used for other interprocess communication, the AST routine does the following:

- Checks for successful completion of the I/O operation by examining the first word in the IOSB
- Checks that the message received is a termination message by examining the message type field in the termination message at the offset ACC$W_MSGTYPE
- Checks for the process identification of the process that has been deleted by examining the second longword of the IOSB
- Checks for the completion status of the process by examining the status field in the termination message at the offset ACC$L_FINALSTS

In this example, the AST service routine performs special action when the subprocess is deleted.

The Create Mailbox and Assign Channel (SYS$CREMBX), Get Device/Volume Information (SYS$GETDVI), and Queue I/O Request (SYS$QIO) system services are described in greater detail in Chapter 9.

4 Using Asynchronous System Traps

This chapter describes the use of asynchronous system traps (ASTs). It contains the following sections:

Section 4.1 provides an overview of AST routines.

Section 4.2 describes access modes for ASTs.

Section 4.3 describes ASTs and process wait states.

Section 4.4 describes how ASTs are declared.

Section 4.5 describes the AST service routine.

Section 4.6 describes how ASTs are delivered, and the affects of kernel threads on AST delivery.

Section 4.7 presents a code example of how to use AST services.

Asynchronous system traps (ASTs) are interrupts that occur asynchronously (out of sequence) with respect to the process's execution. The trap provides a transfer of control to a user-specified procedure that handles the event. For example, you can use them to signal a program to execute a routine whenever a certain condition occurs.

Some system services allow a process to request that it be interrupted when a particular event occurs. Table 4–1 shows the system services that are AST services.

Table 4–1 AST System Services

System Service	Task Performed
SYS$SETAST	Set AST Enable
SYS$DCLAST	Declare AST
SYS$SETPRA	Set Power Recovery AST

The system services that use the AST mechanism accept as an argument the address of an AST service routine, that is, a routine to be given control when the event occurs.

Table 4–2 shows some of the services that use ASTs.

Table 4–2 System Services That Use ASTs

System Service	Task Performed
SYS$DCLAST	Declare AST
SYS$ENQ	Enqueue Lock Request
SYS$GETDVI	Get Device/Volume Information
SYS$GETJPI	Get Job/Process Information
SYS$GETSYI	Get Systemwide Information
SYS$QIO	Queue I/O Request
SYS$SETIMR	Set Timer
SYS$SETPRA	Set Power Recovery AST
SYS$UPDSEC	Update Section File on Disk

For example, if you call the Set Timer (SYS$SETIMR) system service, you can specify the address of a routine to be executed when a time interval expires or at a particular time of day. The service schedules the execution of the routine and returns; the program image continues executing. When the requested timer event occurs, the system "delivers" an AST by interrupting the process and calling the specified routine.

Example 4–1 shows a typical program that calls the SYS$SETIMR system service with a request for an AST when a timer event occurs.

Example 4–1 Calling the SYS$SETIMR System Service

```
#include <stdio.h>
#include <stdlib.h>
#include <ssdef.h>
#include <descrip.h>

struct {
        unsigned int lower, upper;
}daytim;

/* AST routine */
void time_ast(void);

main() {
        unsigned int status;
        $DESCRIPTOR(timbuf,"0 ::10.00"); /* 10-second delta */


/* Convert ASCII format time to binary format */

        status = SYS$BINTIM(&timbuf,    /* buffer containing ASCII time */
                            &daytim);   /* timadr (buffer to receive  */
                                        /* binary time) */
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
        else
                printf("Converting time to binary format...\n");

/* Set the timer */
```

(continued on next page)

Example 4–1 (Cont.) Calling the SYS$SETIMR System Service

```
        status = SYS$SETIMR(0,           /* efn (event flag) */       1
                      &daytim,     /* expiration time */
                      &time_ast,   /* astadr (AST routine) */
                      0,           /* reqidt (timer request id) */
                      0);          /* flags */
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
        else
                printf("Setting the timer to expire in 10 secs...\n"); 2

/* Hibernate the process until the timer expires */

        status = SYS$HIBER();
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

}

void time_ast (void) {

        unsigned int status;

        status = SYS$WAKE(0,    /* process id */
                    0);     /* process name */

        if ((status & 1) != 1)
                LIB$SIGNAL(status);

        printf("Executing AST routine to perform wake up...\n");  3

        return;
}
```

1 The call to the SYS$SETIMR system service requests an AST at 10 seconds from the current time.

The **daytim** argument refers to the quadword, which must contain the time in system time (64-bit) format. For details on how this is accomplished, see Chapter 5. The **astadr** argument refers to TIME_AST, the address of the AST service routine.

When the call to the system service completes, the process continues execution.

2 The timer expires in 10 seconds and notifies the system. The system interrupts execution of the process and gives control to the AST service routine.

3 The user routine TIME_AST handles the interrupt. When the AST routine completes, it issues a RET instruction to return control to the program. The program resumes execution at the point at which it was interrupted.

The following sections describe in more detail how ASTs work and how to use them.

4.1 Overview of AST Routines

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

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

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

The AST routine must observe the following restrictions:

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

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

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

- Event—The following system routines allow you to specify an AST routine to be delivered when the system routine completes:
  - LIB$SPAWN—Signals when the subprocess has been created.
  - SYS$ENQ and SYS$ENQW—Signal when the resource lock is blocking a request from another process.
  - SYS$GETDVI and SYS$GETDVIW—Indicate that device information has been received.
  - SYS$GETJPI and SYS$GETJPIW—Indicate that process information has been received.
  - SYS$GETSYI and SYS$GETSYIW—Indicate that system information has been received.
  - SYS$QIO and SYS$QIOW—Signal when the requested I/O is completed.
  - SYS$UPDSEC—Signals when the section file has been updated.
  - SYS$ABORT_TRANS and SYS$ABORT_TRANSW—Signal when a transaction is aborted.

  - SYS$AUDIT_EVENT and SYS$AUDIT_EVENTW—Signal when an event message is appended to the system security audit log file or send an alarm to a security operator terminal.
  - SYS$BRKTHRU and SYS$BRKTHRU(W)—Signal when a message is sent to one or more terminals.
  - SYS$CHECK_PRIVILEGE and SYS$CHECK_PRIVILEGEW—Signal when the caller has the specified privileges or identifier.
  - VAX SYS$DNS and SYS$DNSW—On VAX systems, signal when client applications are allowed to store resource names and addresses. ♦
  - SYS$END_TRANS and SYS$END_TRANSW—Signal an end to a transaction by attempting to commit it.
  - SYS$GETQUI and SYS$GETQUIW—Signal when information is returned about queues and the jobs initiated from those queues.
  - SYS$START_TRANS and SYS$START_TRANSW—Signal the start of a new transaction.
  - Alpha SYS$SETCLUEVT and SYS$SETCLUEVTW—On Alpha systems, signal a request for notification when a VMScluster configuration event occurs. ♦

- Event—The SYS$SETPRA system service allows you to specify an AST to be delivered when the system detects a power recovery.
- Time—The SYS$SETIMR system service allows you to specify a time for the AST to be delivered.
- Time—The SYS$DCLAST system service delivers a specified AST immediately. This makes it an ideal tool for debugging AST routines.

If a program queues an AST and then exits before the AST is delivered, the AST is deleted from the queue. If a process is hibernating when an AST is delivered, the AST executes and the process continues hibernating.

If a suspended process receives an AST, the execution of the AST depends on the AST mode and the mode at which the process was suspended, as follows:

- If the process was suspended from a SYS$SUSPEND call at supervisor mode, user-mode ASTs are executed as soon as the process is resumed. If more than one AST is delivered, they are executed in the order in which they were delivered. Supervisor-, executive-, and kernel-mode ASTs are executed upon delivery.
- If the process was suspended from a SYS$SUSPEND call at kernel mode, all ASTs are blocked and are executed as soon as the process is resumed.

Generally, AST routines are used with the SYS$QIO or SYS$QIOW system service for handling Ctrl/C, Ctrl/Y, and unsolicited input.

4.2 Access Modes for AST Execution

Each request for an AST is associated with the access mode from which the AST is requested. Thus, if an image executing in user mode requests notification of an event by means of an AST, the AST service routine executes in user mode.

Because the ASTs you use almost always execute in user mode, you do not need to be concerned with access modes. However, you should be aware of some system considerations for AST delivery. These considerations are described in Section 4.6.

4.3 ASTs and Process Wait States

A process in a wait state can be interrupted for the delivery of an AST and the execution of an AST service routine. When the AST service routine completes execution, the process is returned to the wait state, if the condition that caused the wait is still in effect.

With the exception of suspended waits (SUSP) and suspended outswapped waits (SUSPO), any wait states can be interrupted.

4.3.1 Event Flag Waits

If a process is waiting for an event flag and is interrupted by an AST, the wait state is restored following execution of the AST service routine. If the flag is set at completion of the AST service routine (for example, by completion of an I/O operation), then the process continues execution when the AST service routine completes.

Event flags are described in Section 14.6.

4.3.2 Hibernation

A process can place itself in a wait state with the Hibernate (SYS$HIBER) system service. This wait state can be interrupted for the delivery of an AST. When the AST service routine completes execution, the process continues hibernation. The process can, however, “wake” itself in the AST service routine or be awakened by another process or as the result of a timer-scheduled wakeup request. Then, it continues execution when the AST service routine completes.

Process suspension is another form of wait; however, a suspended process cannot be interrupted by an AST. Process hibernation and suspension are described in Chapter 3.

4.3.3 Resource Waits and Page Faults

When a process is executing an image, the system can place the process in a wait state until a required resource becomes available, or until a page in its virtual address space is paged into memory. These waits, which are generally transparent to the process, can also be interrupted for the delivery of an AST.

4.4 How ASTs Are Declared

Most ASTs occur as the result of the completion of an asynchronous event initiated by a system service (for example, a SYS$QIO or SYS$SETIMR request) when the process requests notification by means of an AST.

The Declare AST (SYS$DCLAST) system service creates ASTs. With this service, a process can declare an AST only for the same or for a less privileged access mode.

You may find occasional use for the SYS$DCLAST system service in your programming applications; you may also find the SYS$DCLAST service useful when you want to test an AST service routine.

4.5 The AST Service Routine

An AST service routine must be a separate procedure. The AST must use the standard call procedure, and the routine must return using a RET instruction. If the service routine modifies any registers other than the standard scratch registers, it must set the appropriate bits in the entry mask so that the contents of those registers are saved.

Because you cannot know when the AST service routine will begin executing, you must take care when you write the AST service routine that it does not modify any data or instructions used by the main procedure (unless, of course, that is its function).

On entry to the AST service routine, the arguments shown in Table 4–3 are passed.

Table 4–3 AST Arguments for VAX Systems and Alpha Systems

VAX System Arguments	Alpha System Arguments
AST parameter	AST parameter
R0	R0
R1	R1
PC	PC
PSL	PS

Registers R0 and R1, the program counter (PC), and the processor status longword (PSL) on VAX systems, or processor status (PS) on Alpha systems were saved when the process was interrupted by delivery of the AST.

The AST parameter is an argument passed to the AST service routine so that it can identify the event that caused the AST. When you call a system service requesting an AST, or when you call the SYS$DCLAST system service, you can supply a value for the AST parameter. If you do not specify a value, the parameter defaults to 0.

The following example illustrates an AST service routine. In this example, the ASTs are queued by the SYS$DCLAST system service; the ASTs are delivered to the process immediately so that the service routine is called following each SYS$DCLAST system service call.

```
#include <stdio.h>
#include <ssdef.h>

/* Declare the AST routine */

void astrtn ( int );

main()
{
        unsigned int status, value1=1, value2=2;
        status = SYS$DCLAST(&astrtn,    /* astadr - AST routine */     1
                            value1,     /* astprm - AST parameter */
                            0);         /* acmode */
        if((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
        status = SYS$DCLAST(&astrtn, value2, 0);
        if((status & 1) != 1)
                LIB$SIGNAL( status );

}

void astrtn (int value) {                                  2

/* Evaluate AST parameter */
        switch (value)
        {
                case 1: printf("Executing AST routine with value 1...\n");
                                goto handler_1;
                                break;

                case 2: printf("Executing AST routine with value 2...\n");
                                goto handler_2;
                                break;

                default: printf("Error\n");

        };

/* Handle first AST */

handler_1:
   .
   .
   .
        return;

/* Handle second AST */

handler_2:
   .
   .
   .
        return;
}
```

1 The program CELESTEF calls the SYS$DCLAST AST system service twice to queue ASTs. Both ASTs specify the AST service routine, ASTRTN. However, a different parameter is passed for each call.

2 The first action this AST routine takes is to check the AST parameter so that it can determine if the AST being delivered is the first or second one declared. The value of the AST parameter determines the flow of execution. If a number of different values are determining a number of different paths of execution, Digital recommends that you use the VAX MACRO instruction CASE.

4.6 AST Delivery

This section describes some conditions affecting AST delivery, and the affect of kernel threads on AST delivery.

4.6.1 Conditions Affecting AST Delivery

When a condition causes an AST to be delivered, the system may not be able to deliver the AST to the process immediately. An AST *cannot* be delivered under any of the following conditions:

- An AST service routine is currently executing at the same or at a more privileged access mode.

  Because ASTs are implicitly disabled when an AST service routine executes, one AST routine cannot be interrupted by another AST routine declared for the same access mode. It can, however, be interrupted for an AST declared for a more privileged access mode.

- AST delivery is explicitly disabled for the access mode.

  A process can disable the delivery of AST interrupts with the Set AST Enable (SYS$SETAST) system service. This service may be useful when a program is executing a sequence of instructions that should not be interrupted for the execution of an AST routine.

  Alpha On Alpha systems, SYS$SETAST is often used in a main program that shares data with an AST routine in order to block AST delivery while the program accesses the shared data. ♦

- The process is executing or waiting at an access mode more privileged than that for which the AST is declared.

  For example, if a user-mode AST is declared as the result of a system service but the program is currently executing at a higher access mode (because of another system service call, for example), the AST is not delivered until the program is once again executing in user mode.

If an AST cannot be delivered when the interrupt occurs, the AST is queued until the conditions disabling delivery are removed. Queued ASTs are ordered by the access mode from which they were declared, with those declared from more privileged access modes at the front of the queue. If more than one AST is queued for an access mode, the ASTs are delivered in the order in which they are queued.

4.6.2 Kernel Threads AST Delivery

Alpha With the kernel threads implementation, ASTs are associated with the kernel thread that initiates them, though it is not required that they execute on the thread that initiates them. The use of the kernel thread's PID in the asynchronous system trap control block (ACB) identifies the initiating thread. Associating an ACB with its initiating thread is required; the arrival of an AST is often the event that allows a thread, waiting on a flag or resource, to be made computable.

An AST, for example, may set a flag or make a resource available, and when the AST is completed, the thread continues its execution in non-AST mode and rechecks the wait condition. If the wait condition is satisfied, the thread continues; if not, the thread goes back into the wait queue.

On the other hand, if an AST executes on a kernel thread other than the one that initiated it, then when the AST completes, the kernel thread that initiated the AST must be made computable to ensure that it rechecks a waiting condition that may now be satisfied.

The queuing and delivery mechanisms of ASTs makes a distinction between outer mode ASTs (user and supervisor modes), and inner mode ASTs (executive and kernel modes). This distinction is necessary because of the requirement to synchronize inner mode access.

With the kernel threads implementation, the standard process control block (PCB) AST queues now appear in the kernel thread block (KTB), so that each kernel thread may receive ASTs independently. These queues receive outer mode ASTs, which are delivered on the kernel thread that initiates them. The PCB has a new set of inner mode queues for inner mode ASTs which require the inner mode semaphore. With the creation of multiple kernel threads, inner mode ASTs are inserted in the PCB queues, and are delivered on whichever kernel thread holds the inner mode semaphore. Inner mode ASTs, which are explicitly declared as thread-safe, are inserted in the KTB queues, and are delivered on the kernel thread that initiates them.

If a thread manager declares a user AST callback, then user mode ASTs are delivered to the thread manager. The thread manager then has the responsibilty of determining in what context the AST should be executed.

4.6.2.1 Outer Mode (User and Supervisor) Non-Serial Delivery of ASTs

Before kernel threads, AST routine code of a given mode has always been able to assume the following:

- It would be processed serially. It would not be interrupted or executed concurrently with any other AST of the same mode.
- It would be processed without same-mode, non-AST level code executing concurrently.

Further, before kernel threads, user mode code could safely access data that it knows is only used by other user mode, non-AST level routines without needing any synchronization mechanisms. The underlying assumption is that only one thread of user mode execution exists. If the current code stream is accessing the data, then by implication no other code stream can be accessing it.

After kernel threads, this assumed behavior of AST routines and user mode code is no longer valid. Multiple user-mode, non-AST level code streams can be executing at the same time. The use of any data that can be accessed by multiple user-mode code streams must be modified to become synchronized using the load-locked (LDx_L) and store-conditional (STx_C) instructions, or by using some other synchronization mechanism.

Kernel threads assumes that multiple threads of execution can be active at one time and includes outer mode ASTs. Within any given kernel thread, outer mode ASTs will still be delivered serially. Also, the kernel thread model allows any combination of multiple outer mode threads, or multiple outer mode ASTs. However, outer-mode AST routines, as well as non-AST outer-mode code, has to be aware that any data structure that can be accessed concurrently by outer-mode code, or by any other outer-mode AST must be protected by some form of synchronization.

Before kernel threads, same-mode ASTs executed in the order that they were queued. After kernel threads and within a single kernel thread, that still is true. However, it is not true process-wide. If two ACBs are queued to two different KTBs, whichever is scheduled first executes first. There is no attempt to schedule kernel threads in such a way to correctly order ASTs that have been queued to them. The ASTs execute in any order and can, in fact, execute concurrently.

4.6.2.2 Inner Mode (Executive and Kernel) AST Delivery

Before kernel threads, OpenVMS implemented AST preemptions in inner modes as follows:

- An executive mode AST can preempt non-AST executive mode processing
- A kernel mode AST can preempt non-AST kernel mode processing, or any executive mode processing
- A special kernel mode AST can preempt a normal kernel mode AST, non-AST kernel mode, or any executive mode
- No ASTs can be delivered when interrupt priority level (IPL) is raised to 2 or above. Special kernel mode ASTs execute entirely at IPL 2 or above, which is what prevents other kernel mode ASTs from executing while the special kernel mode AST is active.

After kernel threads, in contrast to the preceeding list, kernel threads delivers any non thread-safe inner mode ASTs to the kernel thread which already owns the semaphore. If no thread currently owns the semaphore when the AST is queued, then the semaphore is acquired in SCH$QAST, and the owner is set to the target kernel thread for that AST. Subsequently queued ASTs see that thread as the semaphore owner and are delivered to that thread. This allows the PALcode and the hardware architecture to process all the AST preemption and ordering rules. ♦

4.7 Example of Using AST Services

The following is an example of a DEC Fortran program that finds the process identification (PID) number of any user working on a particular disk and delivers an AST to notify the user that the disk is coming down:

```
PROGRAM DISK_DOWN
! Implicit none
! Status variable
INTEGER STATUS
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/ DVILIST(2),
2               JPILIST(2)
! Information for GETDVI call
INTEGER PID_BUF,
2       PID_LEN
! Information for GETJPI call
CHARACTER*7 TERM_NAME
INTEGER TERM_LEN
EXTERNAL DVI$_PID,
2        JPI$_TERMINAL
! AST routine and flag
INTEGER AST_FLAG
```

```
PARAMETER (AST_FLAG = 2)
EXTERNAL NOTIFY_USER

INTEGER SYS$GETDVIW,
2       SYS$GETJPI,
2       SYS$WAITFR

! Set up for SYS$GETDVI
DVILIST(1).BUFLEN = 4
DVILIST(1).CODE   = %LOC(DVI$_PID)
DVILIST(1).BUFADR = %LOC(PID_BUF)
DVILIST(1).RETLENADR = %LOC(PID_LEN)
DVILIST(2).END_LIST = 0
! Find PID number of process using SYS$DRIVE0
STATUS = SYS$GETDVIW (,
2                     ,
2                     '_MTA0:',       ! device
2                     DVILIST,        ! item list
2                     ,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Get terminal name and fire AST
JPILIST(1).CODE = %LOC(JPI$_TERMINAL)
JPILIST(1).BUFLEN = 7
JPILIST(1).BUFADR = %LOC(TERM_NAME)
JPILIST(1).RETLENADR = %LOC(TERM_LEN)
JPILIST(2).END_LIST = 0
STATUS = SYS$GETJPI (,
2                    PID_BUF,         !process id
2                    ,
2                    JPILIST,         !itemlist
2                    ,
2                    NOTIFY_USER,     !AST
2                    TERM_NAME)       !AST arg
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Ensure that AST was executed
STATUS = SYS$WAITFR(%VAL(AST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
END


SUBROUTINE NOTIFY_USER (TERM_STR)
! AST routine that broadcasts a message to TERMINAL
! Dummy argument
CHARACTER*(*) TERM_STR
CHARACTER*8 TERMINAL
INTEGER LENGTH
! Status variable
INTEGER STATUS
CHARACTER*(*) MESSAGE
PARAMETER (MESSAGE =
2              'SYS$TAPE going down in 10 minutes')
! Flag to indicate AST executed
INTEGER AST_FLAG

! Declare system routines
INTRINSIC LEN
INTEGER  SYS$BRDCST,
2        SYS$SETEF
EXTERNAL SYS$BRDCST,
2        SYS$SETEF,
2        LIB$SIGNAL
! Add underscore to device name
LENGTH = LEN (TERM_STR)
TERMINAL(2:LENGTH+1) = TERM_STR
TERMINAL(1:1) = '_'
```

```
! Send message
STATUS = SYS$BRDCST(MESSAGE,
2                   TERMINAL(1:LENGTH+1))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
! Set event flag
STATUS = SYS$SETEF (%VAL(AST_FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
END
```

5
System Time Operations

This chapter describes the types of system time operations performed by the operating system. It contains the following sections:

Section 5.1 describes the system time format.

Section 5.2 describes time conversion and date/time manipulation.

Section 5.3 describes how to get the current date and time and set the current time.

Section 5.4 describes how to set and cancel timer requests and how to schedule and cancel wakeups.

Section 5.5 describes using run-time library (RTL) routines to collect timer statistics.

Section 5.6 describes using date/time formatting routines.

Section 5.7 describes the Coordinated Universal Time (UTC) system.

5.1 System Time Format

The operating system maintains the current date and time in 64-bit format. The time value is a binary number in 100-nanosecond (ns) units offset from the system base date and time, which is 00:00 o'clock, November 17, 1858 (the Smithsonian base date and time for the astronomic calendar). Time values must be passed to or returned from system services as the address of a quadword containing the time in 64-bit format. A time value can be expressed as either of the following:

- An absolute time that is a specific date or time of day, or both. Absolute times are always positive values (or 0).
- A delta time that is an offset from the current time to a time or date in the future. Delta times are always expressed as negative values.

If you specify 0 as the address of a time value, the operating system supplies the current date and time.

5.1.1 Absolute Time Format

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

```
dd-MMM-yyyy hh:mm:ss.cc
```

dd	Day of the month (2 characters)
MMM	Month (first 3 characters of the month in uppercase)
yyyy	Year (4 characters)

hh	Hours of the day in 24-hour format (2 characters)
mm	Minutes (2 characters)
ss.cc	Seconds and hundredths of a second (5 characters)

5.1.2 Delta Time Format

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

```
dddd hh:mm:ss.cc
```

dddd	Day of the month (4 characters)
hh	Hour of the day (2 characters)
mm	Minutes (2 characters)
ss.cc	Seconds and hundredths of a second (5 characters)

A delta time is maintained as an integer value representing an amount of time in 100-ns units.

5.2 Time Conversion and Date/Time Manipulation

This section presents information about time conversion and date/time manipulation features, and the routines available to implement them.

5.2.1 Time Conversion Routines

Since the timer system services require you to specify the time in a 64-bit format, you can use time conversion run-time library and system service routines to work with time in a different format. Run-time library and system services do the following:

- Obtain the current date and time in an ASCII string or in system format.
- Convert an ASCII string into the system time format.
- Convert a system time value into an ASCII string.
- Convert the time from system format to integer values.

Table 5–1 shows time conversion run-time and system service routines.

Table 5–1 Time Conversion Routines and System Services

Routine	Function
Time Conversion Run-Time Library (LIB$) Routines	
LIB$CONVERT_DATE_STRING	Converts an input date/time string to an operating system internal time.
LIB$CVT_FROM_INTERNAL_TIME	Converts an operating system standard internal binary time value to an external integer value. The value is converted according to a selected unit of time operation.
LIB$CVTF_FROM_INTERNAL_TIME	Converts an operating system standard internal binary time to an external F-floating point value. The value is converted according to a selected unit of time operation.
LIB$CVT_TO_INTERNAL_TIME	Converts an external integer time value to an operating system standard internal binary time value. The value is converted according to a selected unit of time operation.
LIB$CVTF_TO_INTERNAL_TIME	Converts an F-floating-point time value to an internal binary time value.
LIB$CVT_VECTIM	Converts a seven-word array (as returned by the SYS$NUMTIM system service) to an operating system standard format internal time.
LIB$FORMAT_DATE_TIME	Allows you to select at run time a specific output language and format for a date or time, or both.
LIB$SYS_ASCTIM	Provides a simplified interface between higher-level languages and the $ASCTIM system service.

(continued on next page)

Table 5–1 (Cont.) Time Conversion Routines and System Services

Routine	Function
Time Conversion System Service Routines	
SYS$ASCTIM	Converts an absolute or delta time from 64-bit binary time format to an ASCII string.
SYS$ASCUTC	Converts an absolute time from 128-bit Coordinated Universal Time (UTC) format to an ASCII string.
SYS$BINTIM	Converts an ASCII string to an absolute or delta time value in a binary time format.
SYS$BINUTC	Converts an ASCII string to an absolute time value in the 128-bit UTC format.
SYS$FAO	Converts a binary value into an ASCII character string in decimal, hexadecimal, or octal notation and returns the character string in an output string.
SYS$GETUTC	Returns the current time in 128-bit UTC format.
SYS$NUMTIM	Converts an absolute or delta time from 64-bit system time format to binary integer date and time values.
SYS$NUMUTC	Converts an absolute 128-bit binary time into its numeric components. The numeric components are returned in local time.
SYS$TIMCON	Converts 128-bit UTC to 64-bit system format or 64-bit system format to 128-bit UTC based on the value of the convert flag.

You can use the SYS$GETTIM system service to get the current time in internal format, or you can use SYS$BINTIM to convert a formatted time to an internal time, as shown in Section 5.3.2. You can also use the LIB$DATE_TIME routine to obtain the time, LIB$CVT_FROM_INTERNAL_TIME to convert an internal time to an external time, and LIB$CVT_TO_INTERNAL to convert from an external time to an internal time.

5.2.1.1 Calculating and Displaying Time with SYS$GETTIM and LIB$SUBX

Example 5–1 calculates differences between the current time and a time input in absolute format, and then displays the result as 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 delta time before being displayed. To change an absolute time to a delta time, negate the time array by subtracting it from 0 (specified as an integer array) using the LIB$SUBX routine, which performs subtraction on signed two's complement integers of arbitrary length. For the absolute or delta time format, see Section 5.1.1 and Section 5.1.2.

Example 5–1 Calculating and Displaying the Time

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

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

5.2.1.2 Obtaining Absolute Time with SYS$ASCTIM and SYS$BINTIM

The Convert Binary Time to ASCII String (SYS$ASCTIM)system service is the converse of the Convert ASCII String to Binary Time (SYS$BINTIM) system service. You provide the service with the time in the ASCII format shown in Section 5.3.2. The service then converts the string to a time value in 64-bit format. You can use this returned value as input to a timer scheduling service.

When you specify the ASCII string buffer, you can omit any of the fields, and the service uses the current date or time value for the field. Thus, if you want a timer request to be date independent, you could format the input buffer for the SYS$BINTIM service as shown in the following example. The two hyphens that are normally embedded in the date field must be included, and at least one blank must precede the time field.

```
#include <stdio.h>
#include <descrip.h>

/* Buffer to receive binary time */
struct {
        unsigned int buff1, buff2;
}binary_noon;

main()  {

        unsigned int status;
        $DESCRIPTOR(ascii_noon,"-- 12:00:00.00");  /* noon (absolute time) */

/* Convert time */
        status = SYS$BINTIM(&ascii_noon,        /* timbuf - ASCII time */
                 &binary_noon);                 /* timadr - binary time */

}
```

When the SYS$BINTIM service completes, a 64-bit time value representing “noon today” is returned in the quadword at BINARY_NOON.

5.2.1.3 Obtaining Delta Time with SYS$BINTIM

The SYS$BINTIM system service also converts ASCII strings to delta time values to be used as input to timer services. The buffer for delta time ASCII strings has the following format:

dddd hh:mm:ss.cc

The first field, indicating the number of days, must be specified as 0 if you are specifying a delta time for the current day.

The following example shows how to use the SYS$BINTIM service to obtain a delta time in system format:

```
#include <stdio.h>
#include <descrip.h>

/* Buffer to receive binary time */
struct {
        unsigned int buff1, buff2;
}btenmin;

main()  {

        unsigned int status;
        $DESCRIPTOR(atenmin,"0 00:10:00.00");  /* 10-min delta */

/* Convert time from ASCII to binary */
        status = SYS$BINTIM(&atenmin,   /* timbuf - time in ASCII */
                &btenmin);              /* timadr - binary time */

}
```

If you are programming in VAX MACRO, you can also specify approximate delta time values when you assemble a program, using two MACRO .LONG directives to represent a time value in 100-ns units. The arithmetic is based on the following formula:

```
1 second = 10 million * 100 ns
```

For example, the following statement defines a delta time value of 5 seconds:

```
FIVESEC:  .LONG -10*1000*1000*5,-1 ; Five seconds
```

The value 10 million is expressed as 10*1000*1000 for readability. Note that the delta time value is negative.

If you use this notation, however, you are limited to the maximum number of 100-ns units that can be expressed in a longword. In time values this is slightly more than 7 minutes.

5.2.1.4 Obtaining Numeric and ASCII Time with SYS$NUMTIM

The Convert Binary Time to Numeric Time (SYS$NUMTIM) system service converts a time in the system format into binary integer values. The service returns each of the components of the time (year, month, day, hour, and so on) into a separate word of a 7-word buffer. The SYS$NUMTIM system service and the format of the information returned are described in the *OpenVMS System Services Reference Manual*.

You use the SYS$ASCTIM system service to format the time in ASCII for inclusion in an output string. The SYS$ASCTIM service accepts as an argument the address of a quadword that contains the time in system format and returns the date and time in ASCII format.

If you want to include the date and time in a character string that contains additional data, you can format the output string with the Formatted ASCII Output (SYS$FAO) system service. The SYS$FAO system service converts binary values to ASCII representations, and substitutes the results in character strings according to directives supplied in an input control string. Among these directives are !%T and !%D, which convert a quadword time value to an ASCII string and substitute the result in an output string. For examples of how to do this, see the discussion of $FAO in the *OpenVMS System Services Reference Manual*.

5.2.2 Date/Time Manipulation Routines

The run-time LIB$ facility provides several date/time manipulation routines. These routines let you add, subtract, and multiply dates and times. Use the LIB$ADDX and LIB$SUBX routines to add and subtract times, since the times are defined in integer arrays. Use LIB$ADD_TIMES and LIB$SUB_TIMES to add and subtract two quadword times. When manipulating delta times, remember that they are stored as negative numbers. For example, to add a delta time to an absolute time, you must subtract the delta time from the absolute time. Use LIB$MULT_DELTA_TIME and LIB$MULTF_DELTA_TIME to multiply delta times by scalar and floating scalar.

Table 5–2 lists all the LIB$ routines that perform date/time manipulation.

Table 5–2 Date/Time Manipulation Routines

Routine	Function
LIB$ADD_TIMES	Adds two quadword times
LIB$FORMAT_DATE_TIME	Formats a date and/or time for output
LIB$FREE_DATE_TIME_CONTEXT	Frees the date/time context
LIB$GET_MAXIMUM_DATE_LENGTH	Returns the maximum possible length of an output date/time string
LIB$GET_USERS_LANGUAGE	Returns the user's selected language
LIB$INIT_DATE_TIME_CONTEXT	Initializes the date/time context with a user-specified format
LIB$MULT_DELTA_TIME	Multiplies a delta time value by an integer scalar value
LIB$MULTF_DELTA_TIME	Multiplies a delta time value by an F-floating point scalar value
LIB$SUB_TIMES	Subtracts two quadword times

5.3 Timer Routines Used to Obtain and Set Current Time

This section presents information about getting the current date and time, and setting current time. The run-time library (LIB$) facility provides date/time utility routines for languages that do not have built-in time and date functions. These routines return information about the current date and time or a date/time specified by the user. You can obtain the current time by using the LIB$DATE_TIME routine or by implementing the SYS$GETTIM system service. To set the current time, use the SYS$SETTIME system service.

Table 5–3 describes the date/time routines.

Table 5–3 Timer RTLs and System Services

Routine	Function
Timer Run-Time Library (LIB$) Routines	
LIB$DATE_TIME	Returns, using a string descriptor, the operating system date and time in the semantics of a string that the user provides.

(continued on next page)

Table 5–3 (Cont.) Timer RTLs and System Services

Routine	Function
Timer Run-Time Library (LIB$) Routines	
LIB$DAY	Returns the number of days since the system zero date of November 17, 1858. This routine takes one required argument and two optional arguments: • The address of a longword to contain the number of days since the system zero date (required) • A quadword passed by reference containing a time in system time format to be used instead of the current system time (optional) • A longword integer to contain the number of 10-millisecond units since midnight (optional)
LIB$DAY_OF_WEEK	Returns the numeric day of the week for an input time value. If the input time value is 0, the current day of the week is returned. The days are numbered 1 through 7: Monday is day 1 and Sunday is day 7.
System Service Routine	
SYS$SETIME	Changes the value of or recalibrates the system time.

5.3.1 Obtaining Current Time and Date with LIB$DATE_TIME

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

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

5.3.2 Obtaining Current Time and Date with SYS$GETTIM

You can obtain the current date and time in internal format with the SYS$GETTIM system service. You can convert from internal to character format with the SYS$ASCTIM system service or a directive to the SYS$FAO system service and convert back to internal format with the SYS$BINTIM system service. The Get Time (SYS$GETTIM) system service places the time into a quadword buffer. For example:

```
/* Buffer to receive the binary time */
struct {
        unsigned int buff1, buff2;
}time;

   .
   .
   .
main() {

        unsigned status;
```

This call to SYS$GETTIM returns the current date and time in system format in the quadword buffer TIME.

The Convert Binary Time to ASCII String (SYS$ASCTIM) system service converts a time in system format to an ASCII string and returns the string in a 23-byte buffer. You call the SYS$ASCTIM system service as follows:

```
#include <stdio.h>
#include <descrip.h>

struct {
        unsigned int buff1, buff2;
}time_value;

main() {

        unsigned int status;
        char timestr[23];
        $DESCRIPTOR(atimenow, timestr);

/* Get binary time */
        status = SYS$GETTIM(&time_value);
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

/* Convert binary time to  ASCII */
        status = SYS$ASCTIM(0,            /* timlen - Length of ASCII string */
                            &atimenow,    /* timbuf - ASCII time buffer */
                            &time_value,  /* timadr - Binary time */
                            0);           /* cvtflags - Conversion indicator */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

}
```

Because the address of a 64-bit time value is not supplied, the default value, 0, is used.

The string the service returns has the following format:

dd-MMM-yyyy hh:mm:ss.cc

dd	Day of the month
mmm	Month (a 3-character alphabetic abbreviation)
yyyy	Year
hh:mm:ss.cc	Time in hours, minutes, seconds, and hundredths of a second

5.3.3 Setting the Current Time with SYS$SETIME

The Set System Time (SYS$SETIME) system service allows a user with the operator (OPER) and logical I/O (LOG_IO) privileges to set the current system time. You can specify a new system time (using the **timadr** argument), or you can recalibrate the current system time using the processor's hardware time-of-year clock (omitting the **timadr** argument). If you specify a time, it must be an absolute time value; a delta time (negative) value is invalid.

The system time is set whenever the system is bootstrapped. Normally you do not need to change the system time between system bootstrap operations; however, in certain circumstances you may want to change the system time without rebooting. For example, you might specify a new system time to synchronize two processors, or to adjust for changes between standard time and Daylight Savings Time. Also, you may want to recalibrate the time to ensure that the system time matches the hardware clock time (the hardware clock is more accurate than the system clock).

The DCL command SET TIME calls the SYS$SETIME system service.

If a process issues a delta time request and then the system time is changed, the interval remaining for the request does not change; the request executes after the specified time has elapsed. If a process issues an absolute time request and the system time is changed, the request executes at the specified time, relative to the new system time.

The following example shows the effect of changing the system time on an existing timer request. In this example, two set timer requests are scheduled: one is to execute after a delta time of 5 minutes and the other specifies an absolute time of 9:00.

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <stdlib.h>

void gemini (int x);
unsigned int status;

/* Buffers to receive binary times */
struct {
        unsigned int buff1, buff2;
}abs_binary, delta_binary;

main() {
        $DESCRIPTOR(abs_time,"-- 19:37:00.00");         /* 9 am absolute time */
        $DESCRIPTOR(delta_time,"0 :00:30");             /* 5-min delta time */

        /* Convert ASCII absolute time to binary format */
        status = SYS$BINTIM( &abs_time,         /* ASCII absolute time */
                                &abs_binary);   /* Converted to binary */
        if (status == SS$_NORMAL)
        {
                status = SYS$SETIMR(0,          /* efn - event flag */
                                 &abs_binary,   /* daytim - expiration time */
                                 &gemini,       /* astadr - AST routine */
                                 1,             /* reqidt - timer request id */
                                 0);            /* flags */
                if (status == SS$_NORMAL)
                        printf("Setting system timer A\n");
        }
        else
                LIB$SIGNAL( status );

        /* Convert ASCII delta time to binary format */
        status = SYS$BINTIM( &delta_time,               /* ASCII delta time */
                                &delta_binary);         /* Converted to binary */
        if (status == SS$_NORMAL)
        {
                printf("Converting delta time to binary format\n");
                status = SYS$SETIMR(0,          /* efn - event flag */
                                &delta_binary,  /* daytim - expiration time */
                                &gemini,        /* astadr - AST routine */
                                2,              /* reqidt - timer request id */
                                0);             /* flags */

                if (status == SS$_NORMAL)
                        printf("Setting system timer B\n");
                else
                        LIB$SIGNAL( status );
        }
        else
                        LIB$SIGNAL( status );

        status = SYS$HIBER();
}

void gemini (int  reqidt) {

        unsigned short outlen;
        unsigned int cvtflg=1;
        char timenow[12];
        char fao_str[80];
        $DESCRIPTOR(nowdesc, timenow);
        $DESCRIPTOR(fao_in, "Request ID !UB answered at !AS");
        $DESCRIPTOR(fao_out, fao_str);

/* Returns and converts the current time */
        status = SYS$ASCTIM( 0,         /* timlen - length of ASCII string */
                        &nowdesc,       /* timbuf - receives ASCII string */
                        0,              /* timadr - time value to convert */
                        cvtflg);        /* cvtflg - conversion flags */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

/* Receives the formatted output string */
        status = SYS$FAO(&fao_in,       /* srcstr - control FAO string */
                        &outlen,        /* outlen - length in bytes */
                        &fao_out,       /* outbuf - output buffer */
                        reqidt,         /* p1 - param needed for 1st FAO dir */
                        &nowdesc);      /* p2 - param needed for 2nd FAO dir */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

        status = LIB$PUT_OUTPUT( &fao_out );

        return;

}
```

The following example shows the output received from the preceding program. Assume the program starts execution at 8:45. Seconds later, the system time is set to 9:15. The timer request that specified an absolute time of 9:00 executes immediately, because 9:00 has passed. The request that specified a delta time of 5 minutes times out at 9:20.

```
$ SHOW TIME
   30-DEC-1993 8:45:04.56                         +----------------------+
$ RUN SCORPIO                                     | operator sets system |
<-------------------------------------------------| time to 9:15         |
Request ID number 1 executed at 09:15:00.00       +----------------------+
Request ID number 2 executed at 09:20:00.02
$
```

5.4 Routines Used for Timer Requests

This section presents information about setting and canceling timer requests, and scheduling and canceling wakeups. Since many applications require the scheduling of program activities based on clock time, the operating system allows an image to schedule events for a specific time of day or after a specified time interval. For example, you can use timer system services to schedule, convert, or cancel events. For example, you can use the timer system services to do the following:

- Schedule the setting of an event flag or the queuing of an asynchronous system trap (AST) for the current process, or cancel a pending request that has not yet been processed
- Schedule a wakeup request for a hibernating process, or cancel a pending wakeup request that has not yet been processed
- Set or recalibrate the current system time, if the caller has the proper user privileges

Table 5–4 system services that set, cancel, and schedule timer requests.

Table 5–4 Timer System Services

Timer System Service Routine	Function
SYS$SETIMR	Sets the timer to expire at a specified time. This service sets a per-thread timer.
SYS$CANTIM	Cancels all or a selected subset of the Set Timer requests previously issued by the current image executing in a process. This service cancels all timers associated with the process.
SYS$SCHDWK	Schedules the awakening (restarting) of a kernel thread that has placed itself in a state of hibernation with the Hibernate (SYS$HIBER) service.
SYS$CANWAK	Removes all scheduled wakeup requests for a process from the timer queue, including those made by the caller or by other processes. The Schedule Wakeup ($SCHDWK) service makes scheduled wakeup requests.

5.4.1 Setting Timer Requests with SYS$SETIMR

Timer requests made with the Set Timer (SYS$SETIMR) system service are queued; that is, they are ordered for processing according to their expiration times. The quota for timer queue entries (TQELM quota) controls the number of entries a process can have pending in this timer queue.

When you call the SYS$SETIMR system service, you can specify either an absolute time or a delta time value. Depending on how you want the request processed, you can specify either or both of the following:

- The number of an event flag to be set when the time expires. If you do not specify an event flag, the system sets event flag 0.
- The address of an AST service routine to be executed when the time expires.

Optionally, you can specify a request identification for the timer request. You can use this identification to cancel the request, if necessary. The request identification is also passed as the AST parameter to the AST service routine, if one is specified, so that the AST service routine can identify the timer request.

Example 5–2 and Example 5–3 show timer requests using event flags and ASTs, respectively. Event flags, event flag services, and ASTs are described in more detail in the Chapter 4.

Example 5–2 Setting an Event Flag

```
#include <stdio.h>
#include <ssdef.h>
#include <descrip.h>

/* Buffer to receive binary time */
struct {
        unsigned int buff1, buff2;
}b30sec;

main() {

        unsigned int efn = 4,status;
        $DESCRIPTOR(a30sec,"0 00:00:30.00");

/* Convert time to binary format */
        status = SYS$BINTIM( &a30sec, /* timbuf - ASCII time */
                             &b30sec);/* timadr - binary time */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Converting ASCII to binary time...\n");

/* Set timer to wait */
        status = SYS$SETIMR( efn, /* efn - event flag */
                          &b30sec,/* daytim - binary time */
                          0,      /* astadr - AST routine */
                          0,      /* reqidt - timer request */
                          0);     /* flags */       1
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Request event flag be set in 30 seconds...\n");
```

(continued on next page)

Example 5–2 (Cont.) Setting an Event Flag

```
/* Wait 30 seconds */
        status = SYS$WAITFR( efn );                          2
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Timer expires...\n");

}
```

1 The call to SYS$SETIMR requests that event flag 4 be set in 30 seconds (expressed in the quadword B30SEC).

2 The Wait for Single Event Flag (SYS$WAITFR) system service places the process in a wait state until the event flag is set. When the timer expires, the flag is set and the process continues execution.

Example 5–3 Specifying an AST Service Routine

```
#include <stdio.h>
#include <descrip.h>

#define NOON 12

struct {
        unsigned int buff1, buff2;
}bnoon;

/* Define the AST routine */

void astserv( int );

main() {
        unsigned int status, reqidt=12;
        $DESCRIPTOR(anoon,"-- 12:00:00.00");

/* Convert ASCII time to binary */
        status = SYS$BINTIM(&anoon,     /* timbuf - ASCII time */   1
                            &bnoon);    /* timadr - binary time buffer */
        if((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Converting ASCII to binary...\n");

/* Set timer */
        status = SYS$SETIMR(0,                  /* efn - event flag */ 2
                            &bnoon,             /* daytim - timer expiration */
                            &astserv,           /* astadr - AST routine */
                            reqidt,             /* reqidt - timer request id */
                            0);                 /* cvtflg - conversion flags */
        if((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Setting timer expiration...\n");

        status = SYS$HIBER();

}

void astserv( int astprm ) {                                     3
```

(continued on next page)

Example 5–3 (Cont.) Specifying an AST Service Routine

```
/* Do something if it's a "noon" request */
        if (astprm == NOON)
                printf("This is a noon AST request\n");
        else
                printf("Handling some other request\n");

        status = SYS$SCHDWK(0, /* pidadr - process id */
                        0);/* prcnam - process name */

        return;
}
```

1 The call to SYS$BINTIM converts the ASCII string representing 12:00 noon to format. The value returned in BNOON is used as input to the SYS$SETIMR system service.

2 The AST routine specified in the SYS$SETIMR request will be called when the timer expires, at 12:00 noon. The **reqidt** argument identifies the timer request. (This argument is passed as the AST parameter and is stored at offset 4 in the argument list. See Chapter 4.) The process continues execution; when the timer expires, it is interrupted by the delivery of the AST. Note that if the current time of day is past noon, the timer expires immediately.

3 This AST service routine checks the parameter passed by the **reqidt** argument to determine whether it must service the 12:00 noon timer request or another type of request (identified by a different **reqidt** value). When the AST service routine completes, the process continues execution at the point of interruption.

5.4.2 Canceling a Timer Request with SYS$CANTIM

The Cancel Timer Request (SYS$CANTIM) system service cancels timer requests that have not been processed. The SYS$CANTIM system service removes the entries from the timer queue. Cancellation is based on the request identification given in the timer request. For example, to cancel the request illustrated in Example 5–3, you would use the following call to SYS$CANTIM:

```
        unsigned int status, reqidt=12;
   .
   .
   .
        status = SYS$CANTIM( reqidt, 0);
```

If you assign the same identification to more than one timer request, all requests with that identification are canceled. If you do not specify the **reqidt** argument, all your requests are canceled.

5.4.3 Scheduling Wakeups with SYS$WAKE

Example 5–2 shows a process placing itself in a wait state using the SYS$SETIMR and SYS$WAITFR services. A process can also make itself inactive by hibernating. A process hibernates by issuing the Hibernate (SYS$HIBER) system service. Hibernation is reversed by a wakeup request, which can be put into effect immediately with the SYS$WAKE system service or scheduled with the Schedule Wakeup (SYS$SCHDWK) system service. For more information about the SYS$HIBER and SYS$WAKE system services, see Chapter 3.

The following example shows a process scheduling a wakeup for itself prior to hibernating:

```
#include <stdio.h>
#include <descrip.h>

struct {
        unsigned int buff1, buff2;
}btensec;

main() {

        unsigned int status;
        $DESCRIPTOR(atensec,"0 00:00:10.00");

/* Convert time */
        status = SYS$BINTIM(&atensec, /* timbuf - ASCII time */
                            &btensec);/* timadr - binary time */
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );

/* Schedule wakeup */
        status = SYS$SCHDWK(0, /* pidadr - process id */
                            0, /* prcnam - process name */
                            &btensec, /* daytim - wake up time */
                            0); /* reptim - repeat interval */
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );

/* Sleep ten seconds */
        status = SYS$HIBER();
        if ((status & 1 ) != 1)
                LIB$SIGNAL( status );
}
```

Note that a suitably privileged process can wake or schedule a wakeup request for another process; thus, cooperating processes can synchronize activity using hibernation and scheduled wakeups. Moreover, when you use the SYS$SCHDWK system service in a program, you can specify that the wakeup request be repeated at fixed time intervals. See Chapter 3 for more information on hibernation and wakeup.

5.4.4 Canceling a Scheduled Wakeup with SYS$CANWAK

You can cancel scheduled wakeup requests that are pending but have not yet been processed with the Cancel Wakeup (SYS$CANWAK) system service. This service cancels a wakeup request for a specific kernel thread, if a process ID is specified. If a process name is specified, then the initial thread's wakeup request is canceled.

The following example shows the scheduling of wakeup requests for the process CYGNUS and the subsequent cancellation of the wakeups. The SYS$SCHDWK system service in this example specifies a delta time of 1 minute and an interval time of 1 minute; the wakeup is repeated every minute until the requests are canceled.

```
#include <stdio.h>
#include <descrip.h>

/* Buffer to hold one minute */

struct {
        unsigned int buff1, buff2;
}interval;

main() {
        unsigned int status;
        $DESCRIPTOR(one_min,"0 00:01:00.00");  /* One minute delta */
        $DESCRIPTOR(cygnus, "CYGNUS");          /* Process name */

/* Convert time to binary */
        status = SYS$BINTIM(&one_min,   /* timbuf - ASCII delta time */
                           &interval);  /* timadr - Buffer to hold binary time */
        if((status & 1) != 1)
                LIB$SIGNAL( status );
        else
                printf("Converting time to binary format...\n");

/* Schedule wakeup */
        status = SYS$SCHDWK(0,          /* pidadr - process id */
                            &cygnus,    /* prcnam - process name */
                            &interval,  /* daytim - time to be awakened */
                            &interval); /* reptim - repeat interval */
        if((status & 1) != 1)
                LIB$SIGNAL( status );
        }
        else
                printf("Scheduling wakeup...\n");

        /* Cancel wakeups */
        status = SYS$CANWAK(0,                  /* pidadr - process id */
                            &cygnus);           /* prcnam - process name */

}
```

5.4.5 Executing a Program at Timed Intervals

To execute a program at timed intervals, you can use either the LIB$SPAWN routine or the SYS$CREPRC system service. With LIB$SPAWN, you can create a subprocess that executes a command procedure containing three commands: the DCL command WAIT, the command that invokes the desired program, and a GOTO command that directs control back to the WAIT command. To prevent the parent process from remaining in hibernation until the subprocess executes, you should execute the subprocess concurrently; that is, you should specify CLI$M_NOWAIT.

For more information about using LIB$SPAWN and SYS$CREPRC, see Chapter 3.

5.5 Routines Used for Timer Statistics

This section presents information about the LIB$INIT_TIMER, LIB$SHOW_TIMER, LIB$STAT_TIMER, and LIB$FREE_TIMER routines. By calling these run-time library routines, 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

Following are descriptions of each routine:

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

  You can let the system write the statistics to SYS$OUTPUT (the default), or you can process the statistics with a routine of your own. To process the statistics yourself, specify the name of your routine in the **action-rtn** argument. You can pass one argument to your routine by naming it in the **user-arg** argument. If you use your own routine, it must be written as an integer function and return an error code (return a value of 1 for success). This error code becomes the error code returned by LIB$SHOW_TIMER. Two arguments are passed to your routine: the first is a passed-length character string containing the formatted statistics, and the second is the value of the fourth argument (if any) specified to LIB$SHOW_TIMER.
- LIB$STAT_TIMER—Obtains one of five unformatted statistics. Specify the statistic you want in the **code** argument. Specify a storage area for the statistic in **value**. The **handle-adr** argument must be the same value as you specified for LIB$INIT_TIMER.
- LIB$FREE_TIMER—You should invoke this procedure when you are done with the timer to ensure the modularity of your program. The value in the **handle-adr** argument must be the same as that specified for LIB$INIT_TIMER.

You must invoke LIB$INIT_TIMER to allocate storage for the timer. You should invoke LIB$FREE_TIMER before you exit from your program unit. In between, you can invoke LIB$SHOW_TIMER or LIB$STAT_TIMER, or both, as often as you want. Example 5–4 invokes LIB$SHOW_TIMER and uses a user-written subprogram either to display the statistics or to write them to a file.

Example 5–4 Displaying and Writing Timer Statistics

```
      .
      .
      .
! Timer arguments
INTEGER*4 TIMER_ADDR,
2         TIMER_DATA,
2         TIMER_ROUTINE
EXTERNAL  TIMER_ROUTINE
! Declare library procedures as functions
INTEGER*4 LIB$INIT_TIMER,
2         LIB$SHOW_TIMER
EXTERNAL  LIB$INIT_TIMER,
2         LIB$SHOW_TIMER
```

(continued on next page)

Example 5–4 (Cont.) Displaying and Writing Timer Statistics

```
! 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)
```

(continued on next page)

Example 5–4 (Cont.) Displaying and Writing Timer Statistics

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

5.6 Date/Time Formatting Routines

This section provides information about using date/time formatting routines that allow you to specify input and output formats other than the standard operating system format for dates and times. These include international formats with appropriate language spellings for days and months.

If the desired language is English (the default language) and the desired format is the standard operating system format, then initialization of logical names is not required in order to use the date/time input and output routines. However, if the desired language and format are not the defaults, the system manager (or any user having CMEXEC, SYSNAM, and SYSPRV privileges) must initialize the required logical names.

5.6.1 Performing Date/Time Logical Initialization

Note

You must complete the initialization steps outlined in this section before you can use any of the date/time input and output routines with languages and formats other than the defaults.

As an alternative to the standard operating system format, the command procedure SYS$MANAGER:LIB$DT_STARTUP.COM defines several output formats for dates and times. This command procedure must be executed by the system manager prior to using any of the run-time library date/time routines for input or output formats other than the default. Ideally, this command procedure should be executed from a site-specific startup procedure.

In addition to defining the date/time formats, the LIB$DT_STARTUP.COM command procedure also defines spellings for date and time elements in languages other than English. If different language spellings are required, the system manager must define the logical name SYS$LANGUAGES before invoking LIB$DT_STARTUP.COM. The translation of SYS$LANGUAGES is then used to select which languages are defined.

Table 5–5 shows the available languages and their logical names.

Table 5–5 Available Languages for Date/Time Formatting

Language	Logical Name
Austrian	AUSTRIAN
Danish	DANISH
Dutch	DUTCH
Finnish	FINNISH
French	FRENCH
French Canadian	CANADIAN
German	GERMAN
Hebrew	HEBREW
Italian	ITALIAN
Norwegian	NORWEGIAN
Portuguese	PORTUGUESE
Spanish	SPANISH
Swedish	SWEDISH
Swiss French	SWISS_FRENCH
Swiss German	SWISS_GERMAN

For example, if the system manager wants the spellings for French, German, and Italian languages to be defined, he or she must define SYS$LANGUAGES as shown, prior to invoking LIB$DT_STARTUP.COM:

```
$ DEFINE SYS$LANGUAGES FRENCH, GERMAN, ITALIAN
```

If the user requires an additional language, for example FINNISH, then the system manager must add FINNISH to the definition of SYS$LANGUAGES and reexecute the command procedure.

Date/Time Manipulation Option

The Date/Time Manipulation option provides date/time spelling support for four new languages. Users or application programmers can select the desired language by defining the logical name SYS$LANGUAGES. The new languages and their equivalent names are as follows:

Language	Equivalent Name
Chinese (simplified character)	Hanzi
Chinese (traditional character)	Hanyu
Korean	Hangul
Thai	Thai

Defining Date/Time Spelling

To define the spelling for Hanzi and Hanyu, define SYS$LANGUAGES as shown below, prior to invoking LIB$DT_STARTUP.COM:

```
$ DEFINE SYS$LANGUAGES HANZI, HANYU
$ @SYS$MANAGER:LIB$DT_STARTUP
```

Predefined Output Formats

Tables 5–6 and 5–7 list the new predefined date and time format logical names, their formats, and examples of the output generated using those formats.

Table 5–6 Predefined Output Date Formats

Logical Name	Format	Example
LIB$DATE_FORMAT_042	!Y4年!MNB月!DB日 !WAU	1994年3月7日 (一)
LIB$DATE_FORMAT_043	!Y4年!MNB月!DB日 !WU	1994年3月7日 星期一
LIB$DATE_FORMAT_044	!Y4年!MNB月!DB日 !WAU	1994年3月7日 (一)
LIB$DATE_FORMAT_045	!Y4年!MNB月!DB日 !WU	1994年3月7日 星期一
LIB$DATE_FORMAT_046	!Y4 년 !MNB 월 !DB 일 !WAU	1994 년 3 월 7 일 (월)
LIB$DATE_FORMAT_047	!Y4 년 !MNB 월 !DB 일 !WU	1994 년 3 월 7 일 월요일

Note

LIB$DATE_FORMAT_042 and LIB$DATE_FORMAT_043 support the DEC Hanzi coded character set.

LIB$DATE_FORMAT_044 and LIB$DATE_FORMAT_045 support the DEC Hanyu coded character set.

LIB$DATE_FORMAT_046 and LIB$DATE_FORMAT_047 support the DEC Hangul coded character set.

Table 5–7 Predefined Output Time Formats

Logical Name	Format	Example
LIB$TIME_FORMAT_021	!MIU!HB2时!MB分!SB秒	上午3时3分6秒
LIB$TIME_FORMAT_022	!MIU!HB2時!MB分!SB秒	上午3時3分6秒
LIB$TIME_FORMAT_023	!MIU !HB2 시 !MB 분 !SB 초	오전 3 시 3 분 6 초

Note

LIB$TIME_FORMAT_021 supports the DEC Hanzi coded character set.

LIB$TIME_FORMAT_022 supports the DEC Hanyu coded character set.

LIB$TIME_FORMAT_023 supports the DEC Hangul coded character set.

Thus, to select a particular format for a date or time, or both, you can define the LIB$DT_FORMAT logical name using the following logicals:

- LIB$DATE_FORMAT_*nnn*, where *nnn* can range from 001 to 047
- LIB$TIME_FORMAT_*nnn*, where *nnn* can range from 001 to 023

5.6.2 Selecting a Format

There are two methods by which date/time input and output formats can be selected:

- The language and format are determined at run time through the translation of the logical names SYS$LANGUAGE, LIB$DT_FORMAT, and LIB$DT_INPUT_FORMAT.
- The language and format are programmable at compile time through the use of the LIB$INIT_DATE_TIME_CONTEXT routine.

In general, if an application accepts text from a user or formats text for presentation to a user, the logical name method of specifying language and format should be used. With this method, the user assigns equivalence names to the logical names SYS$LANGUAGE, LIB$DT_FORMAT, and LIB$DT_INPUT_FORMAT, thereby selecting the language and input or output format of the date and time at run time.

If an application reads text from internal storage or formats text for internal storage or transmission, the language and format should be specified at compile time. If this is the case, the routine LIB$INIT_DATE_TIME_CONTEXT is used to specify the language and format of choice.

5.6.2.1 Formatting Run-Time Mnemonics

The format mnemonics listed in Table 5–8 are used to define both input and output formats at run time.

Table 5–8 Format Mnemonics

Date	Explanation
!D0	Day; zero-filled
!DD	Day; no fill
!DB	Day; blank-filled
!WU	Weekday; uppercase
!WAU	Weekday; abbreviated, uppercase
!WC	Weekday; capitalized
!WAC	Weekday; abbreviated, capitalized
!WL	Weekday; lowercase
!WAL	Weekday; abbreviated, lowercase
!MAU	Month; alphabetic, uppercase
!MAAU	Month; alphabetic, abbreviated, uppercase
!MAC	Month; alphabetic, capitalized
!MAAC	Month; alphabetic, abbreviated, capitalized

(continued on next page)

Table 5–8 (Cont.) Format Mnemonics

Date	Explanation
!MAL	Month; alphabetic, lowercase
!MAAL	Month; alphabetic, abbreviated, lowercase
!MN0	Month; numeric, zero-filled
!MNM	Month; numeric, no fill
!MNB	Month; numeric, blank-filled
!Y4	Year; 4 digits
!Y3	Year; 3 digits
!Y2	Year; 2 digits
!Y1	Year; 1 digit
Time	**Explanation**
!H04	Hours; zero-filled, 24-hour clock
!HH4	Hours; no fill, 24-hour clock
!HB4	Hours; blank-filled, 24-hour clock
!H02	Hours; zero-filled, 12-hour clock
!HH2	Hours; no fill, 12-hour clock
!HB2	Hours; blank-filled, 12-hour clock
!M0	Minutes; zero-filled
!MM	Minutes; no fill
!MB	Minutes; blank-filled
!S0	Seconds; zero-filled
!SS	Seconds; no fill
!SB	Seconds; blank-filled
!C7	Fractional seconds; 7 digits
!C6	Fractional seconds; 6 digits
!C5	Fractional seconds; 5 digits
!C4	Fractional seconds; 4 digits
!C3	Fractional seconds; 3 digits
!C2	Fractional seconds; 2 digits
!C1	Fractional seconds; 1 digit
!MIU	Meridiem indicator; uppercase
!MIC	Meridiem indicator; capitalized (mixed case)
!MIL	Meridiem indicator; lowercase

5.6.2.2 Specifying Formats at Run Time

If an application accepts text from a user or formats text for presentation to a user, the logical name method of specifying language and format should be used. With this method, the user assigns equivalence names to the logical names SYS$LANGUAGE, LIB$DT_FORMAT, and LIB$DT_INPUT_FORMAT, thereby selecting the language and format of the date and time at run time. LIB$DT_INPUT_FORMAT must be defined using the mnemonics listed in Table 5–8. The possible choices for SYS$LANGUAGE and LIB$DT_FORMAT are defined in the

SYS$MANAGER:LIB$DT_STARTUP.COM command procedure that is executed by the system manager prior to using these routines.

The following actions occur when any translation of a logical name fails:

- If the translation of SYS$LANGUAGE or any logical name relating to text fails, then English is used and a status of LIB$_ENGLUSED is returned.
- If the translation of LIBDT_FORMAT, LIBDT_INPUT_FORMAT, or any logical name relating to format fails, the operating system standard (SYS$ASCTIM) representation of the date and time is used, that is, *dd-MMM-yyyy hh:mm:ss.cc*, and a status of LIB$_DEFFORUSE is returned.

Since English is the default language and must therefore always be available, English spellings are not taken from logical name translations, but rather are looked up in an internal table.

5.6.2.3 Specifying Input Formats at Run Time

Using the logical name LIB$DT_INPUT_FORMAT, the user can define his or her own input format at run time using the mnemonics listed in Table 5–8. Once an input format is defined, any dates or times that are input to the application are parsed against this format. For example:

```
$ DEFINE LIB$DT_INPUT_FORMAT -
_$ "!MAU !DD, !Y4 !H02:!M0:!S0:!C2 !MIU"
```

A valid input date string would be as follows:

```
JUNE 15, 1993 08:45:06:50 PM
```

If the user has selected a language other than English, then the translation of SYS$LANGUAGE is used by the parser to recognize alphabetic months and meridiem indicators in the selected language.

Input Format String

The input format string used to define the input date/time format must contain at least the first seven of the following eight fields:

- Month (either alphabetic or numeric)
- Day of the month (numeric)
- Year (from 1 to 4 digits)
- Hour (12- or 24-hour clock)
- Minute of the hour
- Second of the minute
- Fractional seconds
- Meridiem indicator (required for 12-hour clock; illegal for 24-hour clock)

If the input format string specifies a 24-hour clock, the string will contain only the first seven fields in the preceding list. If a 12-hour clock is specified, the eighth field (the meridiem indicator) is required.

The format string fields must appear in two groups: one for date and one for time (date and time fields cannot be intermixed within a group). For the input format, alphabetic case distinctions and abbreviation-specific codes have no significance. For example, the following format string specifies that the month name will be uppercase and spelled out in full:

```
!MAU !DD, !Y4 !H02:!M0:!S0:!C2 !MIU
```

If the input string corresponding to this format string contains a month name that is abbreviated and lowercase, the parse of the input string still works correctly. For example:

```
feb 25, 1988 04:39:02:55 am
```

If this input string is entered, the parse still recognizes “feb” as the month name and “am” as the meridiem indicator, even though the format string specified both of these fields as uppercase, and the month name as unabbreviated.

Punctuation in the Format and Input Strings

One important aspect to consider when formatting date/time input strings is punctuation. The punctuation referred to here is the characters that separate the various date/time fields or the date and time groups. Punctuation in these strings is important because it is used as an outline for the parser, allowing the parser to synchronize the input fields to the format fields.

There are three distinct classes of punctuation:

- None

  Although it is common for no punctuation to begin or end an input format string, you can specify a date/time format that also has no punctuation between the fields or groups of the format string. If this is the case, the corresponding input string must not have any punctuation between the respective fields or groups, although white space (see the next item in this list) may appear at the beginning or end of the input string.

- White space

  White space includes any combination of spaces and tabs. In the interpretation of the format string, any white space is condensed to a single space. When parsing an input string, white space is generally noted as synchronizing punctuation and is skipped; however, white space is significant in some situations, such as with blank-filled numbers.

- Explicit

  Explicit punctuation refers to any string of one or more characters that is used as punctuation and is not solely comprised of white space. Any white space appearing within an explicit punctuation string is interpreted literally; in other words, the white space is not compressed. In the format string, you can use explicit punctuation to denote a particular format and to guide the parser in parsing the input string. In the input string, you can use explicit punctuation to synchronize the parse of the input string against the format string. The explicit punctuation used should not be a subset of the valid input of any field that it precedes or follows it.

Punctuation is especially important in providing guidelines for the parser to translate the input date/time string properly.

Default Date/Time Fields

Punctuation in a date/time string is also useful for specifying which fields you want to omit in order to accept the default values. That is, you can control the parsing of the input string by supplying punctuation without the appropriate field values. If only the punctuation is supplied and a user-supplied default is not specified, the value of the omitted field defaults according to the following rules:

- For the date group, the default is the current date.

- For the time group, the default is 00:00:00.00.

Table 5–9 gives some examples of input strings (using punctuation to indicate defaulted fields) and their full translations (assuming a current date of 25-FEB-1993 and using the default input format).

Table 5–9 Input String Punctuation and Defaults

Input	Full Date/Time Input String
31	31-FEB-1993 00:00:00.00
-MAR	25-MAR-1993 00:00:00.00
-SEPTEMBER	25-SEP-1993 00:00:00.00
-1993	25-FEB-1993 00:00:00.00
23:	25-FEB-1993 23:00:00.00
:45:	25-FEB-1993 00:45:00.00
::23	25-FEB-1993 00:00:23.00
.01	25-FEB-1993 00:00:00.01

Note on the Changing Century

Because the default is the current date for the date group, if you specify a value of 00 with the !Y2 format, the year is interpreted as 1900. After January 1, 2000, the value 00 will be interpreted as 2000.

For example, 02/29/00 is interpreted as 29-FEB-1900, which results in LIB$_INVTIME because 1900 is not a leap year. After the turn of the century (the year 2000), 02/29/00 will be 29-FEB-2000, which is a valid date because 2000 is a leap year.

5.6.2.4 Specifying Output Formats at Run Time

If the logical name method is used to specify an output format at run time, the translations of the logical names SYS$LANGUAGE and LIB$DT_FORMAT specify one or more executive mode logical names which in turn must be translated to determine the actual format string. These additional logical names supply such things as the names of the days of the week and the months in the selected language (as determined by SYS$LANGUAGE). All of these logicals are predefined, so that a nonprivileged user can select any one of these languages and formats. In addition, a user can create his or her own languages and formats; however, the CMEXEC, SYSNAM and SYSPRV privileges are required.

To select a particular format for a date or time, or both, you must define the LIB$DT_FORMAT logical name using the following:

- LIB$DATE_FORMAT_*nnn*, where *nnn* ranges from 001 to 040
- LIB$TIME_FORMAT_*nnn*, where *nnn* ranges from 001 to 020

The order in which these logical names appear in the definition of LIB$DT_FORMAT determines the order in which they are output. A single space is inserted into the output string between the two elements, if the definition specifies that both are output. For example:

```
$ DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_006, LIB$TIME_FORMAT_012
```

This definition causes the date to be output in the specified format, followed by a space and the time in the specified format, as follows:

```
13 JAN 93 9:13 AM
```

Table 5–10 lists all predefined date format logical names, their formats, and examples of the output generated using those formats. (The mnemonics used to specify the formats are listed in Table 5–8.)

Table 5–10 Predefined Output Date Formats

Date Format Logical Name	Format	Example
LIB$DATE_FORMAT_001	!DB-!MAAU-!Y4	13-JAN-1993
LIB$DATE_FORMAT_002	!DB !MAU !Y4	13 JANUARY 1993
LIB$DATE_FORMAT_003	!DB.!MAU !Y4	13.JANUARY 1993
LIB$DATE_FORMAT_004	!DB.!MAU.!Y4	13.JANUARY.1993
LIB$DATE_FORMAT_005	!DB !MAU !Y2	13 JANUARY 93
LIB$DATE_FORMAT_006	!DB !MAAU !Y2	13 JAN 93
LIB$DATE_FORMAT_007	!DB.!MAAU !Y2	13.JAN 93
LIB$DATE_FORMAT_008	!DB.!MAAU.!Y2	13.JAN.93
LIB$DATE_FORMAT_009	!DB !MAAU !Y4	13 JAN 1993
LIB$DATE_FORMAT_010	!DB.!MAAU !Y4	13.JAN 1993
LIB$DATE_FORMAT_011	!DB.!MAAU.!Y4	13.JAN.1993
LIB$DATE_FORMAT_012	!MAU !DD, !Y4	JANUARY 13, 1993
LIB$DATE_FORMAT_013	!MN0/!D0/!Y2	01/13/93
LIB$DATE_FORMAT_014	!MN0-!D0-!Y2	01-13-93
LIB$DATE_FORMAT_015	!MN0.!D0.!Y2	01.13.93
LIB$DATE_FORMAT_016	!MN0 !D0 !Y2	01 13 93
LIB$DATE_FORMAT_017	!D0/!MN0/!Y2	13/01/93
LIB$DATE_FORMAT_018	!D0/!MN0-!Y2	13/01-93
LIB$DATE_FORMAT_019	!D0-!MN0-!Y2	13-01-93
LIB$DATE_FORMAT_020	!D0.!MN0.!Y2	13.01.93
LIB$DATE_FORMAT_021	!D0 !MN0 !Y2	13 01 93
LIB$DATE_FORMAT_022	!Y2/!MN0/!D0	93/01/13
LIB$DATE_FORMAT_023	!Y2-!MN0-!D0	93-01-13
LIB$DATE_FORMAT_024	!Y2.!MN0.!D0	93.01.13
LIB$DATE_FORMAT_025	!Y2 !MN0 !D0	93 01 13
LIB$DATE_FORMAT_026	!Y2!MN0!D0	930113
LIB$DATE_FORMAT_027	/!Y2.!MN0.!D0	/93.01.13
LIB$DATE_FORMAT_028	!MN0/!D0/!Y4	01/13/1993
LIB$DATE_FORMAT_029	!MN0-!D0-!Y4	01-13-1993
LIB$DATE_FORMAT_030	!MN0.!D0.!Y4	01.13.1993
LIB$DATE_FORMAT_031	!MN0 !D0 !Y4	01 13 1993
LIB$DATE_FORMAT_032	!D0/!MN0/!Y4	13/01/1993

(continued on next page)

Table 5–10 (Cont.) Predefined Output Date Formats

Date Format Logical Name	Format	Example
LIB$DATE_FORMAT_033	!D0-!MN0-!Y4	13-01-1993
LIB$DATE_FORMAT_034	!D0.!MN0.!Y4	13.01.1993
LIB$DATE_FORMAT_035	!D0 !MN0 !Y4	13 01 1993
LIB$DATE_FORMAT_036	!Y4/!MN0/!D0	1993/01/13
LIB$DATE_FORMAT_037	!Y4-!MN0-!D0	1993-01-13
LIB$DATE_FORMAT_038	!Y4.!MN0.!D0	1993.01.13
LIB$DATE_FORMAT_039	!Y4 !MN0 !D0	1993 01 13
LIB$DATE_FORMAT_040	!Y4!MN0!D0	19930113

Table 5–11 lists all predefined time format logical names, their formats, and examples of the output generated using those formats.

Table 5–11 Predefined Output Time Formats

Time Format Logical	Format	Example
LIB$TIME_FORMAT_001	!H04:!M0:!S0.!C2	09:13:25.14
LIB$TIME_FORMAT_002	!H04:!M0:!S0	09:13:25
LIB$TIME_FORMAT_003	!H04.!M0.!S0	09.13.25
LIB$TIME_FORMAT_004	!H04 !M0 !S0	09 13 25
LIB$TIME_FORMAT_005	!H04:!M0	09:13
LIB$TIME_FORMAT_006	!H04.!M0	09.13
LIB$TIME_FORMAT_007	!H04 !M0	09 13
LIB$TIME_FORMAT_008	!HH4:!M0	9:13
LIB$TIME_FORMAT_009	!HH4.!M0	9.13
LIB$TIME_FORMAT_010	!HH4 !M0	9 13
LIB$TIME_FORMAT_011	!H02:!M0 !MIU	09:13 AM
LIB$TIME_FORMAT_012	!HH2:!M0 !MIU	9:13 AM
LIB$TIME_FORMAT_013	!H04!M0	0913
LIB$TIME_FORMAT_014	!H04H!M0m	09H13m
LIB$TIME_FORMAT_015	kl !H04.!M0	kl 09.13
LIB$TIME_FORMAT_016	!H04H!M0'	09H13'
LIB$TIME_FORMAT_017	!H04.!M0 h	09.13 h
LIB$TIME_FORMAT_018	h !H04.!M0	h 09.13
LIB$TIME_FORMAT_019	!HH4 h !MM	9 h 13
LIB$TIME_FORMAT_020	!HH4 h !MM min !SS s	9 h 13 min 25 s

5.6.2.5 Specifying Formats at Compile Time

If an application reads text from internal storage or formats text for internal storage or transmission, the language and format should be specified at compile time. The routine LIB$INIT_DATE_TIME_CONTEXT allows the user to specify the language and format at compile time by initializing the context area used by LIB$FORMAT_DATE_TIME for output or LIB$CONVERT_DATE_STRING for input with specific strings, instead of through logical name translations. Note that when the text will be parsed by another program, LIB$INIT_DATE_TIME_CONTEXT expects all required context information (including spellings) to be specified. For applications where the context specifies a user's preferred format style, the spellings can be looked up from the logical name tables.

Only one context component can be initialized per call to LIB$INIT_DATE_TIME_CONTEXT. Table 5–12 lists the available components and their number of elements. (_ABB indicates an abbreviated version of the month and weekday names.)

Table 5–12 Available Components for Specifying Formats at Compile Time

Available Component	Number of Elements
LIB$K_MONTH_NAME	12
LIB$K_MONTH_NAME_ABB	12
LIB$K_FORMAT_MNEMONICS	9
LIB$K_WEEKDAY_NAME	7
LIB$K_WEEKDAY_NAME_ABB	7
LIB$K_RELATIVE_DAY_NAME	3
LIB$K_MERIDIEM_INDICATOR	2
LIB$K_OUTPUT_FORMAT	2
LIB$K_INPUT_FORMAT	1
LIB$K_LANGUAGE	1

To specify the actual values for these elements, you must use an initialization string in the following format:

"[delim][string-1][delim][string-2][delim]...[delim][string-n][delim]"

In this format, [-] is a delimiting character that is not in any of the strings, and [*string-n*] is the spelling of the *n*th instance of the component.

For example, a string passed to this routine to specify the English spellings of the abbreviated month names might be as follows:

" | JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC | "

The string must contain the exact number of elements for the associated component; otherwise the error LIB$_NUMELEMENTS is returned. Note that the string begins and ends with a delimiter. Thus, there is one more delimiter than the number of string elements in the initialization string.

5.6.2.6 Specifying Input Format Mnemonics at Compile Time

To specify the input format mnemonics at compile time, the user must initialize the component LIB$K_FORMAT_MNEMONICS with the appropriate values. Table 5–13 lists the nine fields that must be initialized, in the appropriate order, along with their default (English) values.

Table 5–13 Legible Format Mnemonics

Order	Format Field	Legible Mnemonic (Default)
1	Year	YYYY
2	Numeric month	MM
3	Numeric day	DD
4	Hours (12- or 24-hour)	HH
5	Minutes	MM
6	Seconds	SS
7	Fractional seconds	CC

(continued on next page)

Table 5–13 (Cont.) Legible Format Mnemonics

Order	Format Field	Legible Mnemonic (Default)
8	Meridiem indicator	AM/PM
9	Alphabetic month	MONTH

For example, the following is a valid definition of the component LIB$K_FORMAT_MNEMONICS, using English as the natural language:

```
|YYYY|MM|DD|HH|MM|SS|CC|AM/PM|MONTH|
```

If the user were entering the same string using Austrian as the natural language, the definition of the component LIB$K_FORMAT_MNEMONICS would be as follows:

```
|JJJJ|MM|TT|SS|MM|SS|HH| |MONAT|
```

5.6.2.7 Specifying Output Formats at Compile Time

To specify an output format at compile time, the user must preinitialize the component LIB$K_OUTPUT_FORMAT. Two elements are associated with this output format string. One describes the date format fields, the other the time format fields. The order in which they appear in the string determines the order in which they are output. A single space is inserted into the output stream between the two elements, if the call to LIB$FORMAT_DATE_TIME specifies that both be output. For example:

" | !DB-!MAAU-!Y4 | !H04:!M0:!S0.!C2 | "

(These mnemonics are listed in Table 5–8.) This format string represents the format used by the $ASCTIM system service for outputting times. Note that the middle delimiter is replaced by a space in the resultant output.

```
13-JAN-1993 14:54:09:24
```

5.6.3 Converting with the LIB$CONVERT_DATE_STRING Routine

The LIB$CONVERT_DATE_STRING routine converts an absolute date/time string into an operating system internal format date/time quadword. You can optionally specify which fields of the input string can be defaulted (using the **input-flags** argument), and what the default values should be (using the **defaults** argument). By default, the time fields can be defaulted but the date fields cannot. Table 5–9 gives some examples of these default values.

The optional **defaulted-fields** argument to LIB$CONVERT_DATE_STRING can be used to determine which input fields were defaulted. That is, the **defaulted-fields** argument is a bit mask in which each set bit indicates that the corresponding field was defaulted in the input date/time string.

If you want to use LIB$CONVERT_DATE_STRING to return the current time as well as the current date, you can call the $NUMTIM system service and pass the **timbuf** argument, which contains the current date and time, to LIB$CONVERT_DATE_STRING as the **defaults** argument. This tells the LIB$CONVERT_DATE_STRING routine to take the default values for the date and time fields from the 7-word array returned by $NUMTIM.

5.6.4 Retrieving with LIB$GET_DATE_FORMAT Routine

The LIB$GET_DATE_FORMAT routine enables you to retrieve information about the currently selected input format. The string returned by LIB$GET_DATE_FORMAT parallels the currently defined input format string, consisting of the format punctuation (with most white space compressed) and legible mnemonics representing the various format fields.

Based on the currently defined input date/time format, LIB$GET_DATE_FORMAT returns a string comprised of the mnemonics that represent the current format. These mnemonics are listed in Table 5–13.

Table 5–14 gives some examples of input format strings and their resultant mnemonic strings (using English as the default language).

Table 5–14 Sample Input Format Strings

Sample Format String	LIB$GET_DATE_FORMAT Value
!MAU !DD, !Y4 !H04:!M0:!S0:!C2	MONTH DD, YYYY4 HH:MM:SS:CC2
!MN0-!D0-!Y2 !H04:!M0:!S0.!C2	MM-DD-YYYY2 HH:MM:SS.CC2
!MN0/!D0/!Y2 !H02:!M0:!S0.!C2 !MIU	MM/DD/YYYY2 HH:MM:SS.CC2 AM/PM

5.6.4.1 Using User-Defined Output Formats

In addition to the 40 date output formats and 20 time output formats provided, users can define their own date and time output formats using the logical names LIB$DATE_FORMAT_*nnn* and LIB$TIME_FORMAT_*nnn*, where *nnn* ranges from 501 to 999. (That is, values of *nnn* from 001 to 500 are reserved for use by Digital Equipment Corporation.) The mnemonics used to define output formats are listed in Table 5–8.

User-defined output formats must be defined as executive-mode logicals, and they must be defined in the table LNM$DT_FORMAT_TABLE. These formats are normally defined from the site-specific startup command procedure. The following example illustrates the steps required of the system manager to create a particular output format using French as the language:

```
$ DEFINE/EXEC/TABLE=LNM$DT_FORMAT_TABLE LIB$DATE_FORMAT_501 -
_$  "!WL, le !DD !MAL !Y4"
$ DEFINE/EXEC/TABLE=LNM$DT_FORMAT_TABLE LIB$TIME_FORMAT_501 -
_$  "!H04 heures et !M0 minutes"
```

After the system manager defines the desired formats, the user can access them by using the following commands:

```
$ DEFINE SYS$LANGUAGE FRENCH
$ DEFINE LIB$DT_FORMAT LIB$DATE_FORMAT_501, LIB$TIME_FORMAT_501
```

After completing these steps, a program outputting the date and time provides the following results:

```
mardi, le 20 janvier 1993 13 heures et 50 minutes
```

In addition to creating their own date and time formats, users can also define their own language tables (provided they have the SYSNAM, SYSPRV and CMEXEC privileges). To create a language table, a user must define all the logical names required.

The following example defines a portion of the Dutch language table. This table is included in its entirety in the set of predefined languages provided with the international date/time formatting routines.

```
$ CREATE/NAME/PARENT=LNM$SYSTEM_DIRECTORY/EXEC/PROT=(S:RWED,G:R,W:R) -
_$  LNM$LANGUAGE_DUTCH
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_DUTCH LIB$WEEKDAYS_L -
_$  "maandag", "dinsdag", "woensdag", "donderdag", "vrijdag", -
_$  "zaterdag", "zondag"
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_DUTCH LIB$WEEKDAY_ABBREVIATIONS_L -
_$  "maa", "din", "woe", "don", "vri", "zat", "zon"
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_DUTCH LIB$MONTHS_L "januari", -
_$  "februari", "maart", "april", "mei", "juni", "juli", "augustus", -
_$  "september", "oktober", "november", "december"
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_DUTCH LIB$MONTH_ABBREVIATIONS_L -
_$  "jan", "feb", "mrt", "apr", "mei", "jun", "jul", "aug", "sep", -
_$  "okt", "nov", "dec"
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_AUSTRIAN LIB$RELATIVE_DAYS_L -
_$  "gisteren", "vandaag", "morgen"
```

All logical names that are used to build a language are as follows:

LIB$WEEKDAYS_[U|L|C]
These logical names supply the names of the weekdays, spelled out in full (uppercase, lowercase, or mixed case). Weekdays must be defined in order, starting with Monday.

LIB$WEEKDAY_ABBREVIATIONS_[U|L|C]
These logical names supply the abbreviated names of the weekdays (uppercase, lowercase, or mixed case). Weekday abbreviations must be defined in order, starting with Monday.

LIB$MONTHS_[U|L|C]
These logical names supply the names of the months, spelled out in full (uppercase, lowercase, or mixed case). Months must be defined in order, starting with January.

LIB$MONTH_ABBREVIATIONS_[U|L|C]
These logical names supply the abbreviated names of the months (uppercase, lowercase, or mixed case). Month abbreviations must be defined in order, starting with January.

LIB$MI_[U|L|C]
These logical names supply the spellings for the meridiem indicators (uppercase, lowercase, or mixed case). Meridiem indicators must be defined in order; the first indicator represents the hours 0:00:0.0 to 11:59:59.99, and the second indicator represents the hours 12:00:00.00 to 23:59:59.99.

LIB$RELATIVE_DAYS_[U|L|C]
These logical names supply the spellings for the relative days (uppercase, lowercase, or mixed case). Relative days must be defined in order: yesterday, today, and tomorrow, respectively.

LIB$FORMAT_MNEMONICS
This logical name supplies the abbreviations for the appropriate format mnemonics. That is, the information supplied in this logical name is used to specify a desired input format in the user-defined language. The format mnemonics, along with their English values, are listed in the order in which they must be defined.

1. Year (YYYY)
2. Numeric month (MM)
3. Day of the month (DD)
4. Hour of the day (HH)
5. Minutes of the hour (MM)
6. Seconds of the minute (SS)
7. Parts of the second (CC)
8. Meridiem indicator (AM/PM)
9. Alphabetic month (MONTH)

The English definition of LIB$FORMAT_MNEMONIC is therefore as follows:

```
$ DEFINE/EXEC/TABLE=LNM$LANGUAGE_ENGLISH LIB$FORMAT_MNEMONICS -
_$  "YYYY", "MM", "DD", "HH", "MM", "SS", "CC", "AM/PM ", "MONTH"
```

5.7 Coordinated Universal Time Format

This section provides information about VAX systems that supply system base date and time format other than the Smithsonian base date and time system. The other base date and time format system is the Coordinated Universal Time (UTC) system. UTC time is determined by a network of atomic clocks that are maintained by standard bodies in several countries. Formerly, applications that spanned time zones often used Greenwich Mean Time (GMT) as a time reference.

UTC binary timestamps are opaque octawords of 128-bits that contain several fields. Important fields of the UTC format are an absolute time value, a time differential factor (TDF) that contains the offset of the host node's clock from UTC, and an inaccuracy, or tolerance, that can be applied to the absolute time value. Unlike UTC, the operating system binary date and timestamps in the Smithsonian base date and time format represent only the local time of the host node; they do not contain TDF values or inaccuracy values.

The UTC system services allow applications to gain the benefits of a Coordinated Universal Time reference. The UTC system services enable applications to reference a common time standard independent of the host's location and local date and time value.

By calling the UTC system services, applications can perform the following functions:

- Obtain binary representations of UTC in the binary UTC format
- Convert the binary operating system format date and time to binary UTC-format date and time
- Convert binary UTC-format date and time to the binary operating system date and time
- Convert ASCII-format date and time to binary UTC-format date and time
- Convert binary UTC-format date and time to ASCII-format date and time

System services that implement the UTC-format date and time are:

- SYS$ASCUTC—Convert UTC to ASCII
- SYS$BINUTC—Convert ASCII String to UTC Binary Time

- SYS$GETUTC—Get UTC Time
- SYS$NUMUTC—Convert UTC Time to Numeric Components
- SYS$TIMCON—Time Converter

For specific implementation information about the UTC system services, see the *OpenVMS System Services Reference Manual*.

6 Using Run-Time Library Routines to Access Operating System Components

This chapter describes the run-time library (RTL) routines that allow access to various operating system components. It contains the following sections:

Section 6.1 describes how to use RTL routines to make system services return different types of strings.

Section 6.2 describes how to use RTL routines to provide access to the command language interpreter.

Section 6.3 describes how to use RTL routines to allow high-level language programs to use most VAX machine instructions or their Alpha equivalent.

Section 6.4 describes how to use RTL routines to allocate processwide resources to a single operating system process.

Section 6.5 describes how to use RTL routines to measure performance.

Section 6.6 describes how to use RTL routines to control output formatting.

Section 6.7 describes how to use RTL routines for miscellaneous interface routines.

Run-time library routines allow access to the following operating system components:

- System services
- Command language interpreter
- Some VAX machine instructions

6.1 System Service Access Routines

You can usually call the OpenVMS system services directly from your program. However, system services return only fixed-length strings. In some applications, you may want the result of a system service to be returned as a character array, dynamic string, or variable-length string. For this reason, the RTL provides **jacket** routines for the system services that return strings.

You call jacket routines exactly as you would the corresponding system service, but you can pass an output argument of any valid string class. The routines write the output string using the semantics (fixed, varying, or dynamic) associated with the string's descriptor.

The jacket routines follow the conventions established for all RTL routines, except that the arguments are listed in the order of the arguments for the corresponding system service. Thus, they may not be listed in the standard RTL order (read, modify, write).

For example, the LIB$SYS_ASCTIM routine calls the SYS$ASCTIM system service to convert a binary date and time value to ASCII text. It returns the resulting string using the semantics that the calling program specifies in the destination string argument.

For further information about the operations of the system services, see the *OpenVMS System Services Reference Manual*.

The RTL routines provide access to only the system services that produce output strings, which are listed in Table 6–1. The corresponding RTL routines recognize all VAX string classes.

The RTL does not provide jacket routines for all the system services that accept strings as input. Your program should pass only fixed-length or dynamic input strings to all system services and RTL jacket routines.

Table 6–1 System Service Access Routines

Entry Point	System Service	Function
LIB$SYS_ASCTIM	$ASCTIM	Converts system time in binary form to ASCII text
LIB$SYS_FAO	$FAO	Converts a binary value to ASCII text
LIB$SYS_FAOL	$FAOL	Converts a binary value to ASCII text, using a list argument
LIB$SYS_GETMSG	$GETMSG	Obtains a system or user-defined message text
LIB$SYS_TRNLOG	$TRNLOG	Returns the translation of the specified logical name

6.2 Access to the Command Language Interpreter

Two command language interpreters (CLIs) are available on the operating system: DCL and MCR. The run-time library provides several routines that provide access to the CLI callback facility. These routines allow your program to call the current CLI. In most cases, these routines are called from programs that execute as part of a command procedure. They allow the command procedure and the CLI to exchange information.

These routines call the CLI associated with the current process to perform the specified function. In some cases, however, a CLI is not present. For example, the program may be running directly as a subprocess or as a detached process. If a CLI is not present, these routines return the status LIB$_NOCLI. Therefore, you should be sure that these routines are called when a CLI is active. Table 6–2 lists the RTL routines that access the CLI.

Table 6–2 CLI Access Routines

Entry Point	Function
LIB$GET_FOREIGN	Gets a command line
LIB$DO_COMMAND	Executes a command line after exiting the current program

(continued on next page)

Table 6–2 (Cont.) CLI Access Routines

Entry Point	Function
LIB$RUN_PROGRAM	Runs another program after exiting the current program (chain)
LIB$GET_SYMBOL	Returns the value of a CLI symbol as a string
LIB$DELETE_SYMBOL	Deletes a CLI symbol
LIB$SET_SYMBOL	Defines or redefines a CLI symbol
LIB$DELETE_LOGICAL	Deletes a supervisor-mode process logical name
LIB$SET_LOGICAL	Defines or redefines a supervisor-mode process logical name
LIB$DISABLE_CTRL	Disables CLI interception of control characters
LIB$ENABLE_CTRL	Enables CLI interception of control characters
LIB$ATTACH	Attaches a terminal to another process
LIB$SPAWN	Creates a subprocess of the current process

The following routines execute only when the current CLI is DCL:

LIB$GET_SYMBOL
LIB$SET_SYMBOL
LIB$DELETE_SYMBOL
LIB$DISABLE_CTRL
LIB$ENABLE_CTRL
LIB$SPAWN
LIB$ATTACH

6.2.1 Obtaining the Command Line

The LIB$GET_FOREIGN routine returns the contents of the command line that you use to activate an image. It can be used to give your program access to the qualifiers of a foreign command or to prompt for further command line text.

A **foreign command** is a command that you can define and then use as if it were a DCL or MCR command in order to run a program. When you use the foreign command at command level, the CLI parses the foreign command only and activates the image. It ignores any options or qualifiers that you have defined for the foreign command. Once the CLI has activated the image, the program can call LIB$GET_FOREIGN to obtain and parse the remainder of the command line (after the command itself) for whatever options it may contain.

The *OpenVMS DCL Dictionary* describes how to define a foreign command.

The action of LIB$GET_FOREIGN depends on the environment in which the image is activated:

- If you use a foreign command to invoke the image, you can call LIB$GET_FOREIGN to obtain the command qualifiers following the foreign command. You can also use LIB$GET_FOREIGN to prompt repeatedly for more qualifiers after the command. This technique is illustrated in the following example.
- If the image is in the SYS$SYSTEM: directory, the image can be invoked by the DCL command MCR or by the MCR CLI. In this case, LIB$GET_FOREIGN returns the command line text following the image name.

- If the image is invoked by the DCL command RUN, you can use LIB$GET_FOREIGN to prompt for additional text.
- If the image is not invoked by a foreign command or by MCR, or if there is no information remaining on the command line, and the user-supplied prompt is present, LIB$GET_INPUT is called to prompt for a command line. If the prompt is not present, LIB$GET_FOREIGN returns a zero-length string.

Example

The following PL/I example illustrates the use of the optional **force-prompt** argument to permit repeated calls to LIB$GET_FOREIGN. The command line text is retrieved on the first pass only; after this, the program prompts from SYS$INPUT.

```
EXAMPLE: ROUTINE OPTIONS (MAIN);

%INCLUDE $STSDEF;           /* Status-testing definitions */

DECLARE COMMAND_LINE CHARACTER(80) VARYING,
        PROMPT_FLAG FIXED BINARY(31) INIT(0),
        LIB$GET_FOREIGN ENTRY (CHARACTER(*) VARYING,
                               CHARACTER(*) VARYING,
                               FIXED BINARY(15),
                               FIXED BINARY(31))
          OPTIONS(VARIABLE) RETURNS (FIXED BINARY(31)),
        RMS$_EOF GLOBALREF FIXED BINARY(31) VALUE;

/* Call LIB$GET_FOREIGN repeatedly to obtain and print
   subcommand text. Exit when end-of-file is found. */

DO WHILE ('1'B);                  /* Do while TRUE */
  STS$VALUE = LIB$GET_FOREIGN
               (COMMAND_LINE,'Input: ',,
                PROMPT_FLAG);
  IF STS$SUCCESS THEN
    PUT LIST ('  Command was ',COMMAND_LINE);
  ELSE DO;
    IF STS$VALUE ^= RMS$_EOF THEN
      PUT LIST ('Error encountered');
    RETURN;
    END;
  PUT SKIP;                   /* Skip to next line */
  END;                     /* End of DO WHILE loop */
END;
```

Assuming that this program is present as SYS$SYSTEM:EXAMPLE.EXE, you can define the foreign command EXAMPLE to invoke it, as follows:

```
$ EXAM*PLE :== $EXAMPLE
```

Note the optional use of the asterisk in the symbol name to denote an abbreviated command name. This permits the command name to be abbreviated as EXAM, EXAMP, EXAMPL or to be specified fully as EXAMPLE. See the *OpenVMS DCL Dictionary* for information about abbreviated command names.

Note that the use of the dollar sign ($) before the image name is required in foreign commands.

Now assume that a user runs the image by typing the foreign command and giving “subcommands” that the program displays:

```
$ EXAMP Subcommand 1
  Command was     SUBCOMMAND 1
Input: Subcommand 2
  Command was     SUBCOMMAND 2
Input: ^Z
$
```

In this example, Subcommand 1 was obtained from the command line; the program prompts the user for the second subcommand. The program terminated when the user pressed the Ctrl/Z key sequence (displayed as ^Z) to indicate end-of-file.

6.2.2 Chaining from One Program to Another

The LIB$RUN_PROGRAM routine causes the current image to exit at the point of the call and directs the CLI, if present, to start running another program. If LIB$RUN_PROGRAM executes successfully, control passes to the second program; if not, control passes to the CLI. The calling program cannot regain control. This technique is called **chaining**.

This routine is provided primarily for compatibility with PDP–11 systems, on which chaining is used to extend the address space of a system. Chaining may also be useful in an operating system environment where address space is severely limited and large images are not possible. For example, you can use chaining to perform system generation on a small virtual address space because disk space is lacking.

With LIB$RUN_PROGRAM, the calling program can pass arguments to the next program in the chain only by using the common storage area. One way to do this is to direct the calling program to call LIB$PUT_COMMON in order to pass the information into the common area. The called program then calls LIB$GET_COMMON to retrieve the data.

In general, this practice is not recommended. There is no convenient way to specify the order and type of arguments passed into the common area, so programs that pass arguments in this way must know about the format of the data before it is passed. Fortran COMMON or BASIC MAP/COMMON areas are global OWN storage. When you use this type of storage, it is very difficult to keep your program modular and AST reentrant. Further, you cannot use LIB$RUN_PROGRAM if a CLI is present, as with image subprocesses and detached subprocesses.

Examples

The following PL/I example illustrates the use of LIB$RUN_PROGRAM. It prompts the user for the name of a program to run and calls the RTL routine to execute the specified program.

```
CHAIN:  ROUTINE OPTIONS (MAIN) RETURNS (FIXED BINARY (31));
DECLARE LIB$RUN_PROGRAM ENTRY (CHARACTER (*))  /* Address of string
                                               /* descriptor       */
        RETURNS (FIXED BINARY (31));           /* Return status    */
%INCLUDE $STSDEF;    /* Include definition of return status values  */
DECLARE COMMAND CHARACTER (80);
        GET LIST (COMMAND) OPTIONS (PROMPT('Program to run: '));
        STS$VALUE = LIB$RUN_PROGRAM (COMMAND);
/*
   If the function call is successful, the program will terminate
   here.  Otherwise, return the error status to command level.
*/
        RETURN (STS$VALUE);
END CHAIN;
```

The following COBOL program also demonstrates the use of LIB$RUN_ PROGRAM. When you compile and link these two programs, the first calls LIB$RUN_PROGRAM, which activates the executable image of the second. This call results in the following screen display:

```
THIS MESSAGE DISPLAYED BY PROGRAM PROG2

WHICH WAS RUN BY PROGRAM PROG1

USING LIB$RUN_PROGRAM
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROG1.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01    PROG-NAME    PIC X(9)     VALUE "PROG2.EXE".
01    STAT         PIC 9(9)     COMP.
    88  SUCCESSFUL              VALUE 1.

ROUTINE DIVISION.

001-MAIN.
        CALL "LIB$RUN_PROGRAM"
            USING BY DESCRIPTOR PROG-NAME
            GIVING STAT.
        IF NOT SUCCESSFUL
            DISPLAY "ATTEMPT TO CHAIN UNSUCCESSFUL"
            STOP RUN.

IDENTIFICATION DIVISION.

PROGRAM-ID.  PROG2.

ENVIRONMENT DIVISION.

DATA DIVISION.

ROUTINE DIVISION.


001-MAIN.
        DISPLAY " ".
        DISPLAY "THIS MESSAGE DISPLAYED BY PROGRAM PROG2".
        DISPLAY " ".
        DISPLAY "WHICH WAS RUN BY PROGRAM PROG1".
        DISPLAY " ".
        DISPLAY "USING LIB$RUN_PROGRAM".
        STOP RUN.
```

6.2.3 Executing a CLI Command

The LIB$DO_COMMAND routine stops program execution and directs the CLI to execute a command. The routine's argument is the text of the command line that you want to execute.

This routine is especially useful when you want to execute a CLI command after your program has finished executing. For example, you could set up a series of conditions, each associated with a different command. You could also use the routine to execute a SUBMIT or PRINT command to handle a file that your program creates.

Because of the following restrictions on LIB$DO_COMMAND, you should be careful when you incorporate it in your program.

- After the call to LIB$DO_COMMAND, the current image exits, and control cannot return to it.
- The text of the command is passed to the current CLI. Because you can define your own CLI in addition to DCL and MCR, you must make sure that the command is handled by the intended CLI.
- If the routine is called from a subprocess and a CLI is not associated with that subprocess, the routine executes correctly.

You can also use LIB$DO_COMMAND to execute a DCL command file. To do this, include the at sign (@) along with a command file specification as the input argument to the routine.

Some DCL CLI$ routines perform the functions of LIB$DO_COMMAND. See the *OpenVMS DCL Dictionary* for more information.

Example

The following PL/I example prompts the user for a DCL command to execute after the program exits:

```
EXECUTE: ROUTINE OPTIONS (MAIN) RETURNS (FIXED BINARY (31));

DECLARE LIB$DO_COMMAND ENTRY (CHARACTER (*))  /* Pass DCL command  */
                                              /*  by descriptor    */
         RETURNS (FIXED BINARY (31));         /* Return status     */
%INCLUDE $STSDEF;    /* Include definition of return status values */

DECLARE COMMAND CHARACTER (80);

        GET LIST (COMMAND) OPTIONS (PROMPT('DCL command to execute: '));
        STS$VALUE = LIB$DO_COMMAND (COMMAND);
/*
   If the call to LIB$DO_COMMAND is successful, the program will terminate
   here.  Otherwise, it will return the error status to command level.
*/

        RETURN (STS$VALUE);

END EXECUTE;
```

This example displays the following prompt:

```
DCL command to execute:
```

What you type after this prompt determines the action of LIB$DO_COMMAND. LIB$DO_COMMAND executes any command that is entered as a valid string according to the syntax of PL/I. If the command you enter is incomplete, you are prompted for the rest of the command. For example, if you enter the SHOW command, you receive the following prompt:

```
$_Show what?:
```

6.2.4 Using Symbols and Logical Names

The RTL provides seven routines that give you access to the CLI callback facility. These routines allow a program to “call back” to the CLI to perform functions that normally are performed by CLI commands. These routines perform the following functions:

LIB$GET_SYMBOL	Returns the value of a CLI symbol as a string. Optionally, this routine also returns the length of the returned value and a value indicating whether the symbol was found in the local or global symbol table. This routine executes only when the current CLI is DCL.
LIB$SET_SYMBOL	Causes the CLI to define or redefine a CLI symbol. The optional argument specifies whether the symbol is to be defined in the local or global symbol table; the default is local. This routine executes only when the current CLI is DCL.
LIB$DELETE_SYMBOL	Causes the CLI to delete a symbol. An optional argument specifies the local or global symbol table. If the argument is omitted, the symbol is deleted from the local symbol table. This routine executes only when the current CLI is DCL.
LIB$SET_LOGICAL	Defines or redefines a supervisor-mode process logical name. Supervisor-mode logical names are not deleted when an image exits. This routine is equivalent to the DCL command DEFINE. LIB$SET_LOGICAL allows the calling program to define a supervisor-mode process logical name without itself executing in supervisor mode.
LIB$DELETE_LOGICAL	Deletes a supervisor-mode process logical name. This routine is equivalent to the DCL command DEASSIGN. LIB$DELETE_LOGICAL does not require the calling program to be executing in supervisor mode to delete a supervisor-mode logical name.

For information about using logical names, see Chapter 10.

6.2.5 Disabling and Enabling Control Characters

Two run-time library routines, LIB$ENABLE_CTRL and LIB$DISABLE_CTRL, allow you to call the CLI to enable or disable control characters. These routines take a longword bit mask argument that specifies the control characters to be disabled or enabled. Acceptable values for this argument are LIB$M_CLI_CTRLY and LIB$M_CLI_CTRLT.

LIB$DISABLE_CTRL	Disables CLI interception of control characters. This routine performs the same function as the DCL command SET NOCONTROL=*n*, where *n* is T or Y. It prevents the currently active CLI from intercepting the control character specified during an interactive session. For example, you might use LIB$DISABLE_CTRL to disable CLI interception of Ctrl/Y. Normally, Ctrl/Y interrupts the current command, command procedure, or image. If LIB$DISABLE_CTRL is called with LIB$M_CLI_CTRLY specified as the control character to be disabled, Ctrl/Y is treated like Ctrl/U followed by a carriage return.
LIB$ENABLE_CTRL	Enables CLI interception of control characters. This routine performs the same function as the DCL command SET CONTROL=*n*, where *n* is T or Y. LIB$ENABLE_CTRL restores the normal operation of Ctrl/Y or Ctrl/T.

6.2.6 Creating and Connecting to a Subprocess

You can use LIB$SPAWN and LIB$ATTACH together to spawn a subprocess and attach the terminal to that subprocess. These routines will execute correctly only if the current CLI is DCL. For more information on the SPAWN and ATTACH commands, see the *OpenVMS DCL Dictionary*. For more information on creating processes, see Chapter 1.

LIB$SPAWN	Spawns a subprocess. This routine is equivalent to the DCL command SPAWN. It requests the CLI to spawn a subprocess for executing CLI commands.
LIB$ATTACH	Attaches the terminal to another process. This routine is equivalent to the DCL command ATTACH. It requests the CLI to detach the terminal from the current process and reattach it to a different process.

6.3 Access to VAX Machine Instructions

The VAX instruction set was designed for efficient use by high-level languages and, therefore, contains many functions that are directly useful in your programs. However, some of these functions cannot be used directly by high-level languages.

The run-time library provides routines that allow your high-level language program to use most VAX machine instructions that are otherwise unavailable. On Alpha machines, these routines execute a series of Alpha instructions that emulate the operation of the VAX instructions. In most cases, these routines simply execute the instruction, using the arguments you provide. Some routines that accept string arguments, however, provide some additional functions that make them easier to use.

These routines fall into the following categories:

- Variable-length bit field instruction routines (Section 6.3.1)
- Integer and floating-point instructions (Section 6.3.2)
- Queue instructions (Section 6.3.3)
- Character string instructions (Section 6.3.4)

- Routine call instructions (Section 6.3.5)
- Cyclic redundancy check (CRC) instruction (Section 6.3.5)

The *VAX Architecture Reference Manual* describes the VAX instruction set in detail.

6.3.1 Variable-Length Bit Field Instruction Routines

The variable-length bit field is a VAX data type used to store small integers packed together in a larger data structure. It is often used to store single flag bits.

The run-time library contains five routines for performing operations on variable-length bit fields. These routines give higher-level languages that do not have the inherent ability to manipulate bit fields direct access to the bit field instructions in the VAX instruction set. Further, if a program calls a routine written in a different language to perform some function that also involves bit manipulation, the called routine can include a call to the run-time library to perform the bit manipulation.

Table 6–3 lists the run-time library variable-length bit field routines.

Table 6–3 Variable-Length Bit Field Routines

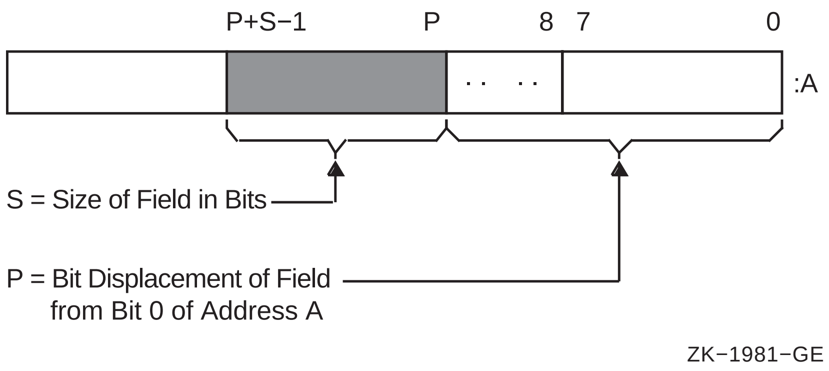
Entry Point	Function
LIB$EXTV	Extracts a field from the specified variable-length bit field and returns it in sign-extended longword form.
LIB$EXTZV	Extracts a field from the specified variable-length bit field and returns it in zero-extended longword form.
LIB$FFC	Searches the specified field for the first clear bit. If it finds one, it returns SS$_NORMAL and the bit position (**find-pos** argument) of the clear bit. If not, it returns a failure status and sets the **find-pos** argument to the start position plus the size.
LIB$FFS	Searches the specified field for the first set bit. If it finds one, it returns SS$_NORMAL and the bit position (**find-pos** argument) of the set bit. If not, it returns a failure status and sets the **find-pos** argument to the start position plus the size.
LIB$INSV	Replaces the specified field with bits 0 through [**size** -1] of the source (**src** argument). If the size argument is 0, nothing is inserted.

Three scalar attributes define a variable bit field:

- Base address—The address of the byte in memory that serves as a reference point for locating the bit field.
- Bit position—The signed longword containing the displacement of the least significant bit of the field with respect to bit 0 of the base address.
- Size—A byte integer indicating the size of the bit field in bits (in the range 0 <= size <= 32). That is, a bit field can be no more than one longword in length.

Figure 6–1 shows the format of a variable-length bit field. The shaded area indicates the field.

Figure 6–1 Format of a Variable-Length Bit Field


P+S−1
P
8
7
0
:A
S = Size of Field in Bits
P = Bit Displacement of Field from Bit 0 of Address A
ZK−1981−GE

Bit fields are zero-origin, which means that the routine regards the first bit in the field as being the zero position. For more detailed information about VAX bit numbering and data formats, see the *VAX Architecture Reference Manual*.

The attributes of the bit field are passed to an RTL routine in the form of three arguments in the following order:

pos

Operating system usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by reference

Bit position relative to the base address. The **pos** argument is the address of a signed longword integer that contains this bit position.

size

Operating system usage: byte_unsigned
type: byte (unsigned)
access: read only
mechanism: by reference

Size of the bit field. The **size** argument is the address of an unsigned byte that contains this size.

base

Operating system usage: longword_unsigned
type: longword (unsigned)
access: read only
mechanism: by reference

Base address. The **base** argument contains the address of the base address.

Example

The following BASIC example illustrates three RTL routines. It opens the terminal as a file and specifies HEX> as the prompt. This prompt allows you to get input from the terminal without the question mark that VAX BASIC normally adds to the prompt in an INPUT statement. The program calls OTS$CVT_TZ_L to convert the character string input to a longword. It then calls LIB$EXTZV once for each position in the longword to extract the bit in that position. Because LIB$EXTVZ is called with a function reference within the PRINT statement, the bits are displayed.

```
10      EXTERNAL LONG FUNCTION
                OTS$CVT_TZ_L,           ! Convert hex text to LONG
                LIB$EXTZV               ! Extract zero-ended bit field

20      OPEN "TT:" FOR INPUT AS FILE #1%     ! Open terminal as a file
        INPUT #1%, "HEX>"; HEXIN$            ! Prompt for input
        STAT%=OTS$CVT_TZ_L(HEXIN$, BINARY%)   ! Convert to longword
        IF (STAT% AND 1%) <> 1%               ! Failed?
        THEN
                PRINT "Conversion failed, decimal status ";STAT%
                GO TO 20                      ! Try again
        ELSE
                PRINT HEXIN$,
                PRINT STR$(LIB$EXTZV(N%, 1%, BINARY%));
                        FOR N%=31% to 0% STEP -1%
```

6.3.2 Integer and Floating-Point Routines

Integer and floating-point routines give a high-level language program access to the corresponding machine instructions. For a complete description of these instructions, see the *VAX Architecture Reference Manual*. Table 6–4 lists the integer and floating-point routines once up front.

Table 6–4 Integer and Floating-Point Routines

Entry Point	Function
LIB$EMUL	Multiplies integers with extended precision
LIB$EDIV	Divides integers with extended precision

6.3.3 Queue Access Routines

A queue is a doubly linked list. A run-time library routine specifies a queue entry by its address. Two longwords, a forward link and a backward link, define the location of the entry in relation to the preceding and succeeding entries. A self-relative queue is a queue in which the links between entries are displacements; the two longwords represent the displacements of the current entry's predecessor and successor. The VAX instructions INSQHI, INSQTI, REMQHI, and REMQTI allow you to insert and remove an entry at the head or tail of a self-relative queue. Each queue instruction has a corresponding RTL routine.

The self-relative queue instructions are interlocked and cannot be interrupted, so that other processes cannot insert or remove queue entries while the current program is doing so. Because the operation requires changing two pointers at the same time, a high-level language cannot perform this operation without calling the RTL queue access routines.

When you use these routines, cooperating processes can communicate without further synchronization and without danger of being interrupted, either on a single processor or in a multiprocessor environment. The queue access routines are also useful in an AST environment; they allow you to add or remove an entry from a queue without being interrupted by an asynchronous system trap.

The remove queue instructions (REMQHI or REMQTI), return the address of the removed entry. Some languages, such as BASIC, COBOL, and Fortran, do not provide a mechanism for accessing an address returned from a routine. Further, BASIC and COBOL do not allow routines to be arguments.

Table 6–5 lists the queue access routines.

Table 6–5 Queue Access Routines

Entry Point	Function
LIB$INSQHI	Inserts queue entry at head
LIB$INSQTI	Inserts queue entry at tail
LIB$REMQHI	Removes queue entry at head
LIB$REMQTI	Removes queue entry at tail

Examples

LIB$INSQHI

In BASIC and Fortran, queues can be quadword aligned in a named COMMON block by using a linker option file to specify alignment of program sections. The LIB$GET_VM routine returns memory that is quadword aligned. Therefore, you should use LIB$GET_VM to allocate the virtual memory for a queue. For instance, to create a COMMON block called QUEUES, use the LINK command with the FILE/OPTIONS qualifier, where FILE.OPT is a linker option file containing the line:

```
PSECT = QUEUES, QUAD
```

A Fortran application using processor-shared memory follows:

```
INTEGER*4 FUNCTION INSERT_Q (QENTRY)
COMMON/QUEUES/QHEADER
INTEGER*4 QENTRY(10), QHEADER(2)
INSERT_Q = LIB$INSQHI (QENTRY, QHEADER)
RETURN
END
```

A BASIC application using processor-shared memory follows:

```
     COM (QUEUES) QENTRY%(9), QHEADER%(1)
     EXTERNAL INTEGER FUNCTION LIB$INSQHI
     IF LIB$INSQHI (QENTRY%() BY REF, QHEADER%() BY REF) AND 1%
          THEN GOTO 1000
             .
             .
             .
1000 REM  INSERTED OK
```

LIB$REMQHI

In Fortran, the address of the removed queue entry can be passed to another routine as an array using the %VAL built-in function.

In the following example, queue entries are 10 longwords, including the two longword pointers at the beginning of each entry:

```
COMMON/QUEUES/QHEADER
INTEGER*4 QHEADER(2), ISTAT
ISTAT = LIB$REMQHI (QHEADER, ADDR)
IF (ISTAT) THEN
        CALL PROC (%VAL (ADDR)) ! Process removed entry
        GO TO ...
ELSE IF (ISTAT .EQ. %LOC(LIB$_QUEWASEMP)) THEN
                GO TO ...       ! Queue was empty
                ELSE IF
                        ...     ! Secondary interlock failed
END IF
   .
   .
   .
END
SUBROUTINE PROC (QENTRY)
INTEGER*4 QENTRY(10)
   .
   .
   .
RETURN
END
```

6.3.4 Character String Routines

The character string routines listed in Table 6–6 give a high-level language program access to the corresponding VAX machine instructions. For a complete description of these instructions, see the *VAX Architecture Reference Manual*. For each instruction, the *VAX Architecture Reference Manual* specifies the contents of all the registers after the instruction executes. The corresponding RTL routines do not make the contents of all the registers available to the calling program.

Table 6–6 lists the LIB$ character string routines and their functions.

Table 6–6 Character String Routines

Entry Point	Function
LIB$LOCC	Locates a character in a string
LIB$MATCHC	Returns the relative position of a substring
LIB$SCANC	Scans characters
LIB$SKPC	Skips characters
LIB$SPANC	Spans characters
LIB$MOVC3	Moves characters
LIB$MOVC5	Moves characters and fills
LIB$MOVTC	Moves translated characters
LIB$MOVTUC	Move translated characters until specified character is found

The *OpenVMS RTL String Manipulation (STR$) Manual* describes STR$ string manipulation routines.

Example

This COBOL program uses LIB$LOCC to return the position of a given letter of the alphabet.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.        LIBLOC.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

01   SEARCH-STRING  PIC X(26)
                    VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
01   SEARCH-CHAR    PIC X.
01   IND-POS        PIC 9(9) USAGE IS COMP.
01   DISP-IND       PIC 9(9).

ROUTINE DIVISION.

001-MAIN.
        MOVE SPACE TO SEARCH-CHAR.
        DISPLAY " ".
        DISPLAY "ENTER SEARCH CHARACTER: " WITH NO ADVANCING.
        ACCEPT SEARCH-CHAR.
        CALL "LIB$LOCC"
            USING BY DESCRIPTOR SEARCH-CHAR, SEARCH-STRING
            GIVING IND-POS.
        IF IND-POS = ZERO
            DISPLAY
                "CHAR ENTERED (" SEARCH-CHAR ") NOT A VALID SEARCH CHAR"
            STOP RUN.
        MOVE IND-POS TO DISP-IND.
        DISPLAY
             "SEARCH CHAR (" SEARCH-CHAR ") WAS FOUND IN POSITION "
             DISP-IND.
        GO TO 001-MAIN.
```

6.3.5 Miscellaneous Instruction Routines

Table 6–7 lists additional routines that you can use.

Table 6–7 Miscellaneous Instruction Routines

Entry Point	Function
LIB$CALLG	Calls a routine using an array argument list
LIB$CRC	Computes a cyclic redundancy check
LIB$CRC_TABLE	Constructs a table for a cyclic redundancy check

LIB$CALLG

The LIB$CALLG routine gives your program access to the CALLG instruction. This instruction calls a routine using an argument list stored as an array in memory, as opposed to the CALLS instruction, in which the argument list is pushed on the stack.

LIB$CRC

The LIB$CRC routine allows your high-level language program to use the CRC instruction, which calculates the cyclic redundancy check. This instruction checks the integrity of a data stream by comparing its state at the sending point and the receiving point. Each character in the data stream is used to generate a value based on a polynomial. The values for each character are then added together. This operation is performed at both ends of the data transmission, and the two result values are compared. If the results disagree, then an error occurred during the transmission.

LIB$CRC_TABLE

The LIB$CRC_TABLE routine takes a polynomial as its input and builds the table that LIB$CRC uses to calculate the CRC. You must specify the polynomial to be used.

For more details, see the *VAX Architecture Reference Manual*.

6.4 Processwide Resource Allocation Routines

This section discusses routines that allocate processwide resources to a single operating system process. The processwide resources discussed here are:

- Local event flags
- BASIC and Fortran logical unit numbers (LUNs)

The resource allocation routines are provided so that user routines can use the processwide resources without conflicting with one another.

In general, you must use run-time library resource allocation routines when your program needs processwide resources. This allows RTL routines, Digital-supplied routines, and user routines that you write to perform together within a process.

If your called routine includes a call to any RTL routine that frees a processwide resource, and that called routine fails to execute normally, the resource will not be freed. Thus, your routine should establish a condition handler that frees the allocated resource before resignaling or unwinding. For information about condition handling, see Chapter 13.

Table 6–8 list routines that perform processwide resource allocation.

Table 6–8 Processwide Resource Allocation Routines

Entry Point	Function
LIB$FREE_LUN	Deallocates a specific logical unit number
LIB$GET_LUN	Allocates next arbitrary logical unit number
LIB$FREE_EF	Frees a local event flag
LIB$GET_EF	Allocates a local event flag
LIB$RESERVE_EF	Reserves a local event flag

6.4.1 Allocating Logical Unit Numbers

BASIC and Fortran use a **logical unit number** (LUN) to define the file or device a program uses to perform input and output. For a routine to be modular, it does not need to know the LUNs being used by other routines that are running at the same time. For this reason, logical units are allocated and deallocated at run time. You can use LIB$GET_LUN and LIB$FREE_LUN to obtain the next available number. This ensures that your BASIC or Fortran routine does not use a logical unit that is already being used by a calling program. Therefore, you should use this routine whenever your program calls or is called by another program that also allocates LUNs. Logical unit numbers 100 to 119 are available to modular routines through these entry points.

To allocate an LUN, call LIB$GET_LUN and use the value returned as the LUN for your I/O statements. If no LUNs are available, an error status is returned and the logical unit is set to −1. When the program unit exits, it should use LIB$FREE_LUN to free any LUNs that have been allocated by LIB$GET_LUN. If it does not free any LUNs, the available pool of numbers is freed for use.

If your called routine contains a call to LIB$FREE_LUN to free the LUNs upon exit, and your routine fails to execute normally, the LUNs will not be freed. For this reason, you should make sure to establish a condition handler to call LIB$FREE_LUN before resignaling or unwinding. Otherwise, the allocated LUN is lost until the image exits.

6.4.2 Allocating Event Flag Numbers

The LIB$GET_EF and LIB$FREE_EF routines operate in a similar way to LIB$GET_LUN and LIB$FREE_LUN. They cause local event flags to be allocated and deallocated at run time, so that your routine remains independent of other routines executing in the same process.

Local event flags numbered 32 to 63 are available to your program. These event flags allow routines to communicate and synchronize their operations. If you use a specific event flag in your routine, another routine may attempt to use the same flag, and the flag will no longer function as expected. Therefore, you should call LIB$GET_EF to obtain the next arbitrary event flag and LIB$FREE_EF to return it to the storage pool. You can obtain a specific event flag number by calling LIB$RESERVE_EF. This routine takes as its argument the event flag number to be allocated.

For information about using event flags, see Chapter 2 and Chapter 14.

6.5 Performance Measurement Routines

The run-time library timing facility consists of four routines to store count and timing information, display the requested information, and deallocate the storage. Table 6–9 lists these routines and their functions.

Table 6–9 Performance Measurement Routines

Entry Point	Function
LIB$INIT_TIMER	Stores the values of the specified times and counts in units of static or heap storage, depending on the value of the routine's argument
LIB$SHOW_TIMER	Gets and formats for output the specified times and counts that are accumulated since the last call to LIB$INIT_TIMER
LIB$STAT_TIMER	Gets one of the times and counts since the last call to LIB$INIT_TIMER and returns it as an unsigned quadword or longword
LIB$FREE_TIMER	Frees the storage allocated by LIB$INIT_TIMER

Using these routines, you can access the following statistics:

- Elapsed time
- CPU time
- Buffered I/O count
- Direct I/O count
- Page faults

The LIB$SHOW_TIMER and LIB$STAT_TIMER routine are relatively simple tools for testing the performance of a new application. To obtain more detailed information, use the system services SYS$GETTIM (Get Time) and SYS$GETJPI (Get Job/Process Information).

The simplest way to use the run-time library routines is to call LIB$INIT_TIMER with no arguments at the beginning of the portion of code to be monitored. This causes the statistics to be placed in OWN storage. To get the statistics from OWN storage, call LIB$SHOW_TIMER (with no arguments) at the end of the portion of code to be monitored.

If you want a particular statistic, you must include a **code** argument with a call to LIB$SHOW_TIMER or LIB$STAT_TIMER. LIB$SHOW_TIMER returns the specified statistic(s) in formatted form and sends them to SYS$OUTPUT. On each call, LIB$STAT_TIMER returns one statistic to the calling program as an unsigned longword or quadword value.

Table 6–10 shows the **code** argument in LIB$SHOW_TIMER or LIB$STAT_TIMER.

Table 6–10 The code Argument in LIB$SHOW_TIMER and LIB$STAT_TIMER

Argument Value	Meaning	LIB$SHOW_TIMER Format	LIB$STAT_TIMER Format
1	Elapsed real time	*dddd hh:mm:ss.cc*	Quadword, in system time format
2	Elapsed CPU time	*hhhh:mm:ss.cc*	Longword, in 10-millisecond increments
3	Number of buffered I/O operations	*nnnn*	Longword
4	Number of direct I/O operations	*nnnn*	Longword
5	Number of page faults	*nnnn*	Longword

When you call LIB$INIT_TIMER, you must use the optional **handler** argument only if you want to keep several sets of statistics simultaneously. This argument points to a block in heap storage where the statistics are to be stored. You need to call LIB$FREE_TIMER only if you have specified **handler** in LIB$INIT_TIMER and you want to deallocate all heap storage resources. In most cases, the implicit deallocation when the image exits is sufficient.

The LIB$STAT_TIMER routine returns only one of the five statistics for each call, and it returns that statistic in the form of an unsigned quadword or longword. LIB$SHOW_TIMER returns the virtual address of the stored information, which BASIC cannot directly access. Therefore, a BASIC program must call LIB$STAT_TIMER and format the returned statistics, as the following example demonstrates.

Example

The following BASIC example uses the run-time library performance analysis routines to obtain timing statistics. It then calls the $ASCTIM system service to translate the 64-bit binary value returned by LIB$STAT_TIMER into an ASCII text string.

```
100    EXTERNAL INTEGER FUNCTION LIB$INIT_TIMER
       EXTERNAL INTEGER FUNCTION LIB$STAT_TIMER
       EXTERNAL INTEGER FUNCTION LIB$FREE_TIMER
       EXTERNAL INTEGER CONSTANT SS$_NORMAL

200    DECLARE LONG COND_VALUE, RANDOM_SLEEP
       DECLARE LONG CODE, HANDLE
       DECLARE STRING TIME_BUFFER
       HANDLE = 0
       TIME_BUFFER = SPACE$(50%)

300    MAP (TIMER) LONG ELAPSED_TIME, FILL
       MAP (TIMER) LONG CPU_TIME
       MAP (TIMER) LONG BUFIO
       MAP (TIMER) LONG DIRIO
       MAP (TIMER) LONG PAGE_FAULTS

400    PRINT "This program returns information about:"
       PRINT "Elapsed time (1)"
       PRINT "CPU time (2)"
       PRINT "Buffered I/O (3)"
       PRINT "Direct I/O (4)"
       PRINT "Page faults (5)"
       PRINT "Enter zero to exit program"
       PRINT "Enter a number from one to"
       PRINT "five for performance information"
       INPUT "One, two, three, four, or five"; CODE
       PRINT

450    GOTO 32766 IF CODE = 0

500    COND_VALUE = LIB$INIT_TIMER( HANDLE )

550    IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
         "Error in initialization"
              GOTO 32767

650    A = 0                !
       FOR I = 1 to 100000  ! This code merely uses some CPU time
       A = A + 1            !
       NEXT I               !

700    COND_VALUE = LIB$STAT_TIMER( CODE, ELAPSED_TIME, HANDLE )

750    IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
         "Error in statistics routine"
              GOTO 32767

800    GOTO 810 IF CODE <> 1%
       CALL SYS$ASCTIM ( , TIME_BUFFER, ELAPSED_TIME, 1% BY VALUE)
       PRINT "Elapsed time: "; TIME_BUFFER
```

```
810    PRINT "CPU time in seconds: "; .01 * CPU_TIME IF CODE = 2%
       PRINT "Buffered I/O: ";BUFIO IF CODE = 3%
       PRINT "Direct I/O: ";DIRIO IF CODE = 4%
       PRINT "Page faults: ";PAGE_FAULTS IF CODE = 5%
       PRINT

900    GOTO 400

32765  COND_VALUE = LIB$FREE_TIMER( HANDLE )
32766  IF (COND_VALUE <> SS$_NORMAL) THEN PRINT @
         "Error in LIB$FREE_TIMER"
                          GOTO 32767

32767  END
```

For information about using system time, see Chapter 5.

6.6 Output Formatting Control Routines

Table 6–11 lists the run-time library routines that customize output.

Table 6–11 Routines for Customizing Output

Entry Point	Function
LIB$CURRENCY	Defines the default currency symbol for process
LIB$DIGIT_SEP	Defines the default digit separator for process
LIB$LP_LINES	Defines the process default size for a printed page
LIB$RADIX_POINT	Defines the process default radix point character

The LIB$CURRENCY, LIB$DIGIT_SEP, LIB$LP_LINES, and LIB$RADIX_POINT routines allow you to customize output. Using them, you can define the logical names SYS$CURRENCY, SYS$DIGIT_SEP, SYS$LP_LINES, and SYS$RADIX_POINT to specify your own currency symbol, digit separator, radix point, or number of lines per printed page. Each routine works by attempting to translate the associated logical name as a process, group, or system logical name. If you have redefined a logical name for a specific local application, then the translation succeeds, and the routine returns the value that corresponds to the option you have chosen. If the translation fails, the routine returns a default value provided by the run-time library, as follows:

$	SYS$CURRENCY
,	SYS$DIGIT_SEP
.	SYS$RADIX_POINT
66	SYS$LP_LINES

For example, if you want to use the British pound sign (£) as the currency symbol within your process, but you want to leave the dollar sign ($) as the system default, define SYS$CURRENCY to be in your process logical name table. Then, any calls to LIB$CURRENCY within your process return “£”, while any calls outside your process return “$”.

You can use LIB$LP_LINES to monitor the current default length of the line printer page. You can also supply your own default length for the current process. United States standard paper size permits 66 lines on each physical page.

If you are writing programs for a utility that formats a listing file to be printed on a line printer, you can use LIB$LP_LINES to make your utility independent of the default page length. Your program can use LIB$LP_LINES to obtain the current length of the page. It can then calculate the number of lines of text per page by subtracting the lines used for margins and headings.

The following is one suggested format:

- Three lines for the top margin
- Three lines for the bottom margin
- Three lines for listing heading information, consisting of:
  - Language-processor identification line
  - Source program identification line
  - One blank line

6.7 Miscellaneous Interface Routines

There are several other RTL routines that permit high-level access to components of the operating system. Table 6–12 lists these routines and their functions. The sections that follow give further details about some of these routines.

Table 6–12 Miscellaneous Interface Routines

Entry Point	Function
LIB$AST_IN_PROG	Indicates whether an asynchronous system trap is in progress
LIB$ASN_WTH_MBX	Assigns an I/O channel and associates it with a mailbox
LIB$CREATE_DIR	Creates a directory or subdirectory
LIB$FIND_IMAGE_SYMBOL	Reads a global symbol from the shareable image file and dynamically activates a shareable image into the P0 address space of a process
LIB$ADDX	Performs addition on signed two's complement integers of arbitrary length (multiple-precision addition)
LIB$SUBX	Performs subtraction on signed two's complement integers of arbitrary length (multiple-precision subtraction)
LIB$FILE_SCAN	Finds file names given OpenVMS RMS file access block (FAB)
LIB$FILE_SCAN_END	End-of-file scan
LIB$FIND_FILE	Finds file names given string
LIB$FIND_FILE_END	End-of-find file

(continued on next page)

Table 6–12 (Cont.) Miscellaneous Interface Routines

Entry Point	Function
LIB$INSERT_TREE	Inserts an element in a binary tree
LIB$LOOKUP_TREE	Finds an element in a binary tree
LIB$TRAVERSE_TREE	Traverses a binary tree
LIB$GET_COMMON	Gets a record from the process's COMMON storage area
LIB$PUT_COMMON	Puts a record to the process's COMMON storage area

6.7.1 Indicating Asynchronous System Trap in Progress

An asynchronous system trap (AST) is a mechanism for providing a software interrupt when an external event occurs, such as when a user presses the Ctrl/C key sequence. When an external event occurs, the operating system interrupts the execution of the current process and calls a routine that you supply. While that routine is active, the AST is said to be in progress, and the process is said to be executing at AST level. When your AST routine returns control to the original process, the AST is no longer active and execution continues where it left off.

The LIB$AST_IN_PROG routine indicates to the calling program whether an AST is currently in progress. Your program can call LIB$AST_IN_PROG to determine whether it is executing at AST level, and then take appropriate action. This routine is useful if you are writing AST-reentrant code.

For information about using ASTs, see Chapter 4.

6.7.2 Create a Directory or Subdirectory

The LIB$CREATE_DIR routine creates a directory or a subdirectory. The calling program must specify the directory specification in standard OpenVMS RMS format. This directory specification may also contain a disk specification.

In addition to the required directory specification argument, LIB$CREATE_DIR takes the following five optional arguments:

- The user identification code (UIC) of the owner of the created directory or subdirectory
- The protection enable mask
- The protection value mask
- The maximum number of versions allowed for files created in this directory or subdirectory
- The relative volume number within the volume set on which the directory or subdirectory is created

See the *OpenVMS RTL Library (LIB$) Manual* for a complete description of LIB$CREATE_DIR.

6.7.3 File Searching Routines

The run-time library provides two routines that your program can call to search for a file and two routines that your program can call to end a search sequence.

- When you call LIB$FILE_SCAN with a wildcard file specification and an action routine, the routine calls the action routine for each file or error, or both, found in the wildcard sequence. LIB$FILE_SCAN allows the search sequence to continue even though certain errors are present.
- When you call LIB$FIND_FILE with a wildcard file specification, it finds the next file specification that matches the wildcard specification.

In addition to the wildcard file specification, which is a required argument, LIB$FIND_FILE takes the following four optional arguments:

- The default specification.
- The related specification.
- The OpenVMS RMS secondary status value from a failing RMS operation.
- A longword containing two flag bits. If bit 1 is set, LIB$FIND_FILE performs temporary defaulting for multiple input files and the related specification argument is ignored. See the *OpenVMS RTL Library (LIB$) Manual* for a complete description of LIB$FIND_FILE in template format.

The LIB$FIND_FILE_END routine is called once after each call to LIB$FIND_FILE in interactive use. LIB$FIND_FILE_END prevents the temporary default values retained by the previous call to LIB$FIND_FILE from affecting the next file specification.

The LIB$FILE_SCAN routine uses an optional context argument to perform temporary defaulting for multiple input files. For example, a command such as the following would specify A, B, and C in successive calls, retaining context, so that portions of one file specification would affect the next file specification:

```
$ COPY  [smith]A,B,C *
```

The LIB$FILE_SCAN_END routine is called once after each sequence of calls to LIB$FILE_SCAN. LIB$FILE_SCAN_END performs a parse of the null string to deallocate saved OpenVMS RMS context and to prevent the temporary default values retained by the previous call to LIB$FILE_SCAN from affecting the next file specification. For instance, in the previous example, LIB$FILE_SCAN_END should be called after the C file specification is parsed, so that specifications from the $COPY files do not affect file specifications in subsequent commands.

The following BLISS example illustrates the use of LIB$FIND_FILE. It prompts for a file specification and default specification. The default specification indicates the default information for the file for which you are searching. Once the routine has searched for one file, the resulting file specification determines both the related file specification and the default file specification for the next search. LIB$FIND_FILE_END is called at the end of the following BLISS program to deallocate the virtual memory used by LIB$FIND_FILE.

```
%TITLE 'FILE_EXAMPLE1 - Sample program using LIB$FIND_FILE'
MODULE FILE_EXAMPLE1(           ! Sample program using LIB$FIND_FILE
                IDENT = '1-001',
                MAIN = EXAMPLE_START
                ) =
BEGIN
```

```
%SBTTL 'Declarations'
!+
! SWITCHES:
!-

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL, NONEXTERNAL = WORD_RELATIVE);

!+
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    EXAMPLE_START;                              ! Main program

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';              ! System symbols

!+
! Define facility-specific messages from shared system messages.
!-
$SHR_MSGDEF(CLI,3,LOCAL,
                (PARSEFAIL,WARNING));
!+
! EXTERNAL REFERENCES:
!-

EXTERNAL ROUTINE
    LIB$GET_INPUT,                             ! Read from SYS$INPUT
    LIB$FIND_FILE,                             ! Wildcard scanning routine
    LIB$FIND_FILE_END,          ! End find file
    LIB$PUT_OUTPUT,                            ! Write to SYS$OUTPUT
    STR$COPY_DX;                               ! String copier

LITERAL
    TRUE = 1,                                  ! Success
    FALSE = 0;                                 ! Failure

%SBTTL 'EXAMPLE_START - Sample program main routine';
ROUTINE EXAMPLE_START =
BEGIN
!+
! This program reads a file specification and default file
! specification from SYS$INPUT.  It then prints all the files that
! match that specification and prompts for another file specification.
! After the first file specification no default specification is requested,
! and the previous resulting file specification becomes the related
! file specification.
!-
LOCAL
    LINEDESC : $BBLOCK[DSC$C_S_BLN],     ! String desc. for input line
    RESULT_DESC : $BBLOCK[DSC$C_S_BLN],  ! String desc. for result file
    CONTEXT,                             ! LIB$FIND_FILE context pointer
    DEFAULT_DESC : $BBLOCK[DSC$C_S_BLN], ! String desc. for default spec
    RELATED_DESC : $BBLOCK[DSC$C_S_BLN], ! String desc. for related spec
    HAVE_DEFAULT,
    STATUS;
!+
! Make all string descriptors dynamic.
!-
CH$FILL(0,DSC$C_S_BLN,LINEDESC);
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;
CH$MOVE(DSC$C_S_BLN,LINEDESC,RESULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,DEFAULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,RELATED_DESC);
HAVE_DEFAULT = FALSE;
CONTEXT = 0;
```

```
!+
! Read file specification, default file specification, and
! related file specification.
!-
WHILE (STATUS = LIB$GET_INPUT(LINEDESC,
                $DESCRIPTOR('FILE SPECIFICATION: '))) NEQ RMS$_EOF
DO BEGIN
    IF NOT .STATUS
        THEN SIGNAL_STOP(.STATUS);
    !+
    ! If default file specification was not obtained, do so now.
    !-
    IF NOT .HAVE_DEFAULT
    THEN BEGIN
        STATUS = LIB$GET_INPUT(DEFAULT_DESC,
                $DESCRIPTOR('DEFAULT FILE SPECIFICATION: '));
        IF NOT .STATUS
            THEN SIGNAL_STOP(.STATUS);
        HAVE_DEFAULT = TRUE;
        END;
    !+
    ! CALL LIB$FIND_FILE until RMS$_NMF (no more files) is returned.
    ! If an error other than RMS$_NMF is returned, it is signaled.
    ! Print out the file specification if the call is successful.
    !-
    WHILE (STATUS = LIB$FIND_FILE(LINEDESC,RESULT_DESC,CONTEXT,
                    DEFAULT_DESC,RELATED_DESC)) NEQ RMS$_NMF
    DO IF NOT .STATUS
        THEN SIGNAL(CLI$_PARSEFAIL,1,RESULT_DESC,.STATUS)
        ELSE LIB$PUT_OUTPUT(RESULT_DESC);
    !+
    ! Make this resultant file specification the related file
    ! specification for next file.
    !-
    STR$COPY_DX(RELATED_DESC,LINEDESC);
    END;                                    ! End of loop
                                            !  reading file specification

!+
! Call LIB$FIND_FILE_END to deallocate the virtual memory used by LIB$FIND_FILE.
! Note that we do this outside of the loop.  Since the MULTIPLE bit of the
! optional user flags argument to LIB$FIND_FILE wasn't used, it is not
! necessary to call LIB$FIND_FILE_END after each call to LIB$FIND_FILE.
! (The MULTIPLE bit would have caused temporary defaulting for multiple input
!  files.)
!-
STATUS = LIB$FIND_FILE_END (CONTEXT);

IF NOT .STATUS
    THEN SIGNAL_STOP (.STATUS);

RETURN TRUE
END;                                        ! End of main program
END                                         ! End of module

ELUDOM
```

The following BLISS example illustrates the use of LIB$FILE_SCAN and LIB$FILE_SCAN_END.

```
%TITLE 'FILE_EXAMPLE2 - Sample program using LIB$FILE_SCAN'
MODULE FILE_EXAMPLE1(            ! Sample program using LIB$FILE_SCAN
        IDENT = '1-001',
        MAIN = EXAMPLE_START
        ) =
BEGIN

%SBTTL 'Declarations'
!+
! SWITCHES:
!-

SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL,
        NONEXTERNAL = WORD_RELATIVE);

!+
! TABLE OF CONTENTS:
!-

FORWARD ROUTINE
    EXAMPLE_START,          ! Main program
    SUCCESS_RTN,            ! Success action routine
    ERROR_RTN;              ! Error action routine

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';      ! System symbols

!+
! Define VMS block structures (BLOCK[,BYTE]).
!-
STRUCTURE
    BBLOCK [O, P, S, E; N] =
                [N]
                (BBLOCK + O) <P, S, E>;
!+
! EXTERNAL REFERENCES:
!-

EXTERNAL ROUTINE
    LIB$GET_INPUT,             ! Read from SYS$INPUT
    LIB$FILE_SCAN,             ! Wildcard scanning routine
    LIB$FILE_SCAN_END,         ! End of file scan
    LIB$PUT_OUTPUT;            ! Write to SYS$OUTPUT
```

```
%SBTTL 'EXAMPLE_START - Sample program main routine';
ROUTINE EXAMPLE_START =
BEGIN
!+
! This program reads the file specification, default file specification,
! and related file specification from SYS$INPUT and then displays on
! SYS$OUTPUT all files which match the specification.
!-
LOCAL
    RESULT_BUFFER : VECTOR[NAM$C_MAXRSS,BYTE], !Buffer for resultant
                                               !  name string
    EXPAND_BUFFER : VECTOR[NAM$C_MAXRSS,BYTE], !Buffer for expanded
                                               !  name string
    LINEDESC : BBLOCK[DSC$C_S_BLN],            !String descriptor
                                               !  for input line
    RESULT_DESC : BBLOCK[DSC$C_S_BLN],         !String descriptor
                                               !  for result file
    DEFAULT_DESC : BBLOCK[DSC$C_S_BLN],        !String descriptor
                                               !  for default specification
    RELATED_DESC : BBLOCK[DSC$C_S_BLN],        !String descriptor
                                               !  for related specification
    IFAB : $FAB_DECL,                          !FAB for file_scan
    INAM : $NAM_DECL,                          !  and a NAM block
    RELNAM : $NAM_DECL,                        !  and a related NAM block
    STATUS;
!+
! Make all descriptors dynamic.
!-
CH$FILL(0,DSC$C_S_BLN,LINEDESC);
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;
CH$MOVE(DSC$C_S_BLN,LINEDESC,RESULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,DEFAULT_DESC);
CH$MOVE(DSC$C_S_BLN,LINEDESC,RELATED_DESC);
!+
! Read file specification, default file specification, and related
! file specification
!-
STATUS = LIB$GET_INPUT(LINEDESC,
                $DESCRIPTOR('File specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
STATUS = LIB$GET_INPUT(DEFAULT_DESC,
                $DESCRIPTOR('Default file specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
STATUS = LIB$GET_INPUT(RELATED_DESC,
                $DESCRIPTOR('Related file specification: '));
IF NOT .STATUS
    THEN SIGNAL_STOP(.STATUS);
!+
! Initialize the FAB, NAM, and related NAM blocks.
!-
$FAB_INIT(FAB=IFAB,
        FNS=.LINEDESC[DSC$W_LENGTH],
        FNA=.LINEDESC[DSC$A_POINTER],
        DNS=.DEFAULT_DESC[DSC$W_LENGTH],
        DNA=.DEFAULT_DESC[DSC$A_POINTER],
        NAM=INAM);

$NAM_INIT(NAM=INAM,
        RSS=NAM$C_MAXRSS,
        RSA=RESULT_BUFFER,
        ESS=NAM$C_MAXRSS,
        ESA=EXPAND_BUFFER,
        RLF=RELNAM);
```

```
$NAM_INIT(NAM=RELNAM);
RELNAM[NAM$B_RSL] = .RELATED_DESC[DSC$W_LENGTH];
RELNAM[NAM$L_RSA] = .RELATED_DESC[DSC$A_POINTER];
!+
! Call LIB$FILE_SCAN.  Note that errors need not be checked
! here because LIB$FILE_SCAN calls error_rtn for all errors.
!-
LIB$FILE_SCAN(IFAB,SUCCESS_RTN,ERROR_RTN);

!+
! Call LIB$FILE_SCAN_END to deallocate virtual memory used for
! file scan structures.
!-
STATUS = LIB$FILE_SCAN_END (IFAB);

IF NOT .STATUS
    THEN SIGNAL_STOP (.STATUS);

RETURN 1
END;                                                ! End of main program

ROUTINE SUCCESS_RTN (IFAB : REF BBLOCK) =
BEGIN
!+
! This routine is called by LIB$FILE_SCAN for each file that it
! successfully finds in the search sequence.
!
! Inputs:
!
!       IFAB    Address of a fab
!
! Outputs:
!
!       file specification printed on SYS$OUTPUT
!-
LOCAL
    DESC : BBLOCK[DSC$C_S_BLN];    ! A local string descriptor
BIND
    INAM = .IFAB[FAB$L_NAM] : BBLOCK;    ! Find NAM block
                                         !   from pointer in FAB
CH$FILL(0,DSC$C_S_BLN,DESC);             ! Make static
                                         !   string descriptor
DESC[DSC$W_LENGTH] = .INAM[NAM$B_RSL];   ! Get string length
                                         !   from NAM block
DESC[DSC$A_POINTER] = .INAM[NAM$L_RSA];  ! Get pointer to the string
RETURN LIB$PUT_OUTPUT(DESC)              ! Print name on SYS$OUTPUT
                                         !   and return
END;

ROUTINE ERROR_RTN (IFAB : REF BBLOCK) =
BEGIN
!+
! This routine is called by LIB$FILE_SCAN for each file specification that
! produces an error.
!
! Inputs:
!
!       ifab     Address of a fab
!
! Outputs:
!
!       Error message is signaled
!-
LOCAL
    DESC : BBLOCK[DSC$C_S_BLN];            ! A local string descriptor
```

```
BIND
    INAM = .IFAB[FAB$L_NAM] : BBLOCK;      ! Get NAM block pointer
                                           !   from FAB

CH$FILL(0,DSC$C_S_BLN,DESC);               ! Create static
                                           !   string descriptor
DESC[DSC$W_LENGTH] = .INAM[NAM$B_RSL];
DESC[DSC$A_POINTER] = .INAM[NAM$L_RSA];
!+
! Signal the error using the shared message PARSEFAIL
! and the CLI facility code.  The second part of the SIGNAL
! is the RMS STS and STV error codes.
!-
RETURN SIGNAL((SHR$_PARSEFAIL+3^16),1,DESC,
                .IFAB[FAB$L_STS],.IFAB[FAB$L_STV])

END;
END                   ! End of module

ELUDOM
```

6.7.4 Inserting an Entry into a Balanced Binary Tree

Three routines allow you to manipulate the contents of a balanced binary tree:

- LIB$INSERT_TREE adds an entry to a balanced binary tree.
- LIB$LOOKUP_TREE looks up an entry in a balanced binary tree.
- LIB$TRAVERSE_TREE calls an action routine for each node in the tree.

Example

The following BLISS example illustrates all three routines. The program prompts for input from SYS$INPUT and stores each data line as an entry in a binary tree. When the user enters the end-of-file character (Ctrl/Z), the tree is printed in sorted order. The program includes three subroutines:

- The first subroutine allocates virtual memory for a node.
- The second subroutine routine compares a key with a node.
- The third subroutine is called during the tree traversal. It prints out the left and right subtree pointers, the current node balance, and the name of the node.

```
%TITLE 'TREE_EXAMPLE   - Sample program using binary tree routines'
MODULE TREE_EXAMPLE(                      ! Sample program using trees
                IDENT = '1-001',
                MAIN = TREE_START
                ) =
BEGIN

%SBTTL 'Declarations'
!+
! SWITCHES:
!-
SWITCHES ADDRESSING_MODE (EXTERNAL = GENERAL, NONEXTERNAL = WORD_RELATIVE);

!+
! LINKAGES:
!
!     NONE
!
! TABLE OF CONTENTS:
!-
```

```
FORWARD ROUTINE
    TREE_START,                         ! Main program
    ALLOC_NODE,                         ! Allocate memory for a node
    COMPARE_NODE,                       ! Compare two nodes
    PRINT_NODE;                         ! Print a node (action routine
                                        !  for LIB$TRAVERSE_TREE)

!+
! INCLUDE FILES:
!-

LIBRARY 'SYS$LIBRARY:STARLET.L32';             ! System symbols

!+
! Define VMS block structures (BLOCK[,BYTE]).
!-
STRUCTURE
    BBLOCK [O, P, S, E; N] =
                [N]
                (BBLOCK + O) <P, S, E>;
!+
! MACROS:
!-
MACRO
    NODE$L_LEFT = 0,0,32,0%,             ! Left subtree pointer in node
    NODE$L_RIGHT = 4,0,32,0%,            ! Right subtree pointer
    NODE$W_BAL = 8,0,16,0%,              ! Balance this node
    NODE$B_NAMLNG = 10,0,8,0%,           ! Length of name in this node
    NODE$T_NAME = 11,0,0,0%;             ! Start of name (variable length)

LITERAL
    NODE$C_LENGTH = 11;                  ! Length of fixed part of node

!+
! EXTERNAL REFERENCES:
!-

EXTERNAL ROUTINE
    LIB$GET_INPUT,                       ! Read from SYS$INPUT
    LIB$GET_VM,                          ! Allocate virtual memory
    LIB$INSERT_TREE,                     ! Insert into binary tree
    LIB$LOOKUP_TREE,                     ! Lookup in binary tree
    LIB$PUT_OUTPUT,                      ! Write to SYS$OUTPUT
    LIB$TRAVERSE_TREE,                   ! Traverse a binary tree
    STR$UPCASE,                          ! Convert string to all uppercase
    SYS$FAO;                             ! Formatted ASCII output routine

%SBTTL 'TREE_START - Sample program main routine';
ROUTINE TREE_START =
BEGIN
!+
! This program reads from SYS$INPUT and stores each data line
! as an entry in a binary tree.  When end-of-file character (CTRL/Z)
! is entered, the tree will be printed in sorted order.
!-
LOCAL
    NODE : REF BBLOCK,                  ! Address of allocated node
    TREEHEAD,                           ! List head of binary tree
    LINEDESC : BBLOCK[DSC$C_S_BLN],     ! String descriptor for input line
    STATUS;
```

```
TREEHEAD = 0;                           ! Zero binary tree head
CH$FILL(0,DSC$C_S_BLN,LINEDESC);        ! Make a dynamic descriptor
LINEDESC[DSC$B_CLASS] = DSC$K_CLASS_D;  ! ...
!+
! Read input lines until end of file seen.
!-
WHILE (STATUS = LIB$GET_INPUT(LINEDESC,           ! Read input line
                        $DESCRIPTOR('Text: ')))   !  with this prompt
                NEQ RMS$_EOF
DO IF NOT .STATUS                            ! Report any errors found
        THEN SIGNAL(.STATUS)
        ELSE BEGIN
            STR$UPCASE(LINEDESC,LINEDESC);   ! Convert string
                                             !  to uppercase
            IF NOT (STATUS = LIB$INSERT_TREE(
                        TREEHEAD,       ! Insert good data into the tree
                        LINEDESC,       ! Data to insert
                        %REF(1),        ! Insert duplicate entries
                        COMPARE_NODE,   ! Addr. of compare routine
                        ALLOC_NODE,     ! Addr. of node allocation routine
                        NODE,           ! Return addr. of
                        0))             !   allocated node here
                THEN SIGNAL(.STATUS);
            END;
!+
! End of file character encountered.  Print the whole tree and exit.
!-
IF NOT (STATUS = LIB$TRAVERSE_TREE(
                        TREEHEAD,       ! Listhead of tree
                        PRINT_NODE,     ! Action routine to print a node
                        0))
    THEN SIGNAL(.STATUS);

RETURN SS$_NORMAL
END;                                     ! End of routine tree_start

ROUTINE ALLOC_NODE (KEYDESC,RETDESC,CONTEXT) =
BEGIN
!+
! This routine allocates virtual memory for a node.
!
! INPUTS:
!
!      KEYDESC                Address of string descriptor for key
!                              (this is the linedesc argument passed
!                              to LIB$INSERT_TREE)
!      RETDESC                Address of location to return address of
!                              allocated memory
!      CONTEXT                Address of user context argument passed
!                              to LIB$INSERT_TREE (not used in this
!                              example)
!
! OUTPUTS:
!
!        Memory address returned in longword pointed to by retdesc
!-
MAP
    KEYDESC : REF BBLOCK,
    RETDESC : REF VECTOR[,LONG];

LOCAL
    NODE : REF BBLOCK,
    STATUS;
```

```
STATUS = LIB$GET_VM(%REF(NODE$C_LENGTH+.KEYDESC[DSC$W_LENGTH]),NODE);
IF NOT .STATUS
    THEN RETURN .STATUS
    ELSE BEGIN
        NODE[NODE$B_NAMLNG] = .KEYDESC[DSC$W_LENGTH];  ! Set name length
        CH$MOVE(.KEYDESC[DSC$W_LENGTH],                ! Copy in the name
                .KEYDESC[DSC$A_POINTER],
                NODE[NODE$T_NAME]);
        RETDESC[0] = .NODE;                    ! Return address to caller
        END;
RETURN .STATUS

END;


ROUTINE COMPARE_NODE (KEYDESC,NODE,CONTEXT) =
BEGIN
!+
! This routine compares a key with a node.
!
! INPUTS:
!
!       KEYDESC           Address of string descriptor for new key
!                          (This is the linedesc argument passed to
!                          LIB$INSERT_TREE)
!       NODE              Address of current node
!       CONTEXT           User context data (Not used in this example)
!-
MAP
    KEYDESC : REF BBLOCK,
    NODE : REF BBLOCK;

RETURN CH$COMPARE(.KEYDESC[DSC$W_LENGTH],          ! Compare key with
                                                   !  current node
                        .KEYDESC[DSC$A_POINTER],
                        .NODE[NODE$B_NAMLNG],
                        NODE[NODE$T_NAME])

END;

ROUTINE PRINT_NODE (NODE,CONTEXT) =
BEGIN
!+
! This routine is called during the tree traversal.  It
! prints out the left and right subtree pointers, the
! current node balance, and the name of the node.
!-
MAP
    NODE : REF BBLOCK;
```

```
LOCAL
    OUTBUF : BBLOCK[512],                   ! FAO output buffer
    OUTDESC : BBLOCK[DSC$C_S_BLN],          ! Output buffer descriptor
    STATUS;
CH$FILL(0,DSC$C_S_BLN,OUTDESC);             ! Zero descriptor
OUTDESC[DSC$W_LENGTH] = 512;
OUTDESC[DSC$A_POINTER] = OUTBUF;
IF NOT (STATUS = SYS$FAO($DESCRIPTOR('!XL !XL !XL !XW !AC'),
                        OUTDESC,OUTDESC,
                        .NODE,.NODE[NODE$L_LEFT],
                        .NODE[NODE$L_RIGHT],
                        .NODE[NODE$W_BAL],
                        NODE[NODE$B_NAMLNG]))
    THEN SIGNAL(.STATUS)
    ELSE BEGIN
        STATUS = LIB$PUT_OUTPUT(OUTDESC);      ! Output the line
        IF NOT .STATUS
            THEN SIGNAL(.STATUS);
        END;

RETURN SS$_NORMAL

END;
END                                       ! End of module TREE_EXAMPLE

ELUDOM
```

7
Run-Time Library Input/Output Operations

This chapter describes the different I/O programming capabilities provided by the run-time library and illustrates these capabilities with examples of common I/O tasks. This chapter contains the following sections:

Section 7.1 describes the input and output operations within a program.

Section 7.2 describes using SYS$INPUT and SYS$OUTPUT.

Section 7.3 describes using LIB$GET_INPUT and LIB$PUT_OUTPUT for simple user I/O.

Section 7.4 describes using the SMG$ run-time library routines for managing the appearance of terminal screens.

Section 7.5 describes using screen management input routines and the SYS$QIO and SYS$QIOW system services to perform special actions.

7.1 Choosing I/O Techniques

The operating system and its compilers provide the following methods for completing input and output operations within a program:

- DEC Text Processing Utility
- DECforms software
- Program language I/O statements
- OpenVMS Record Management Services (RMS) and Run-Time Library (RTL) routines
- SYS$QIO and SYS$QIOW system services
- Non-Digital-supplied device drivers to control the I/O to the device itself

The DEC Text Processing Utility (DECTPU) is a text processor that can be used to create text editing interfaces. DECTPU has the following features:

- High-level procedure language with several data types, relational operators, error interception, looping, case language statements, and built-in procedures
- Compiler for the DECTPU procedure language
- Interpreter for the DECTPU procedure language
- Extensible Versatile Editor (EVE) editing interface which, in addition to the EVE keypad, provides EDT, VT100, WPS, and numeric keypad emulation

In addition, DECTPU offers the following special features:

- Multiple buffers
- Multiple windows
- Multiple subprocesses

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

The method you select for I/O operations depends on the task you want to accomplish, ease of use, speed, and level of control you want.

The DECforms software is a forms management product for transaction processing. DECforms integrates text and graphics into forms and menus that application programs use as an interface to users. DECforms software offers application developers software development tools and a run-time environment for implementing interfaces.

DECforms software integrates with the Application Control and Management System (ACMS), a transaction process (TP) monitor that works with other Digital commercial applications to provide complete customizable development and run-time environments for TP applications. An asynchronous call interface to ACMS allows a single DECforms run-time process to control multiple terminals simultaneously in a multithreaded way, resulting in an efficient use of memory. By using the ACMS Remote Access Option, DECforms software can be distributed to remote CPUs. This technique allows the host CPU to offload forms processing and distribute it as closely as possible to the end user.

In contrast to OpenVMS RMS, RTLs, SYS$QIOs, and device driver I/O, program language I/O statements have the slowest speed and lowest level of control, but they are the easiest to use and are highly portable.

OpenVMS RMS and RTL routines can perform most I/O operations for a high-level or assembly language program. For information about OpenVMS RMS, see the *OpenVMS Record Management Services Reference Manual*.

System services can complete any I/O operation and can access devices not supported within OpenVMS RMS. See Chapter 9 for a description of using I/O system services.

Writing a device driver provides the most control over I/O operations, but can be more complex to implement. For information about device drivers for VAX systems, see the *OpenVMS VAX Device Support Manual*.

Several types of I/O operations can be performed within a program, including the following:

- RTL routines allow you to read simple input from a user or send simple output to a user. One RTL routine allows you to specify a string to prompt for input from the current input device, defined by SYS$INPUT. Another RTL routine allows you to write a string to the current output device, defined by SYS$OUTPUT. See Section 7.2 and Section 7.3 for more information.
- RTL routines allow you to read complex input from a user or to send complex output to a user. By providing an extensive number of screen management (SMG$) routines, the RTL allows you to read multiple lines of input from users or to send complex output to users. The SMG$ routines also allow you to create and modify complicated displays that accept input and produce output. See Section 7.4 for more information.

- RTL routines allow you to use programming language I/O statements to send data to and receive data from files. Program language I/O statements call OpenVMS RMS routines to complete most file I/O. You can also use OpenVMS RMS directly in your programs for accomplishing file I/O. See Chapter 8 for more information.
- The SYS$QIO and SYS$QIOW system services allow you to send data to and from devices with the most flexibility and control. You can use system services to access devices not supported by your programming language or by OpenVMS RMS.

  You can perform other special I/O actions, such as interrupts, controlling echo, handling unsolicited input, using the type-ahead buffer, using case conversion, and sending sytem broadcast messges, by using SMG$ routines or, for example, by using SYS$BRKTHRU system service to broadcast messages. See Section 7.5 for more information.

7.2 Using SYS$INPUT and SYS$OUTPUT

Typically, you set up your program so that the user is the invoker. The user starts the program by entering a DCL command associated with the program or by using the RUN command.

7.2.1 Default Input and Output Devices

The user's input and output devices are defined by the logical names SYS$INPUT and SYS$OUTPUT, which are initially set to the values listed in Table 7–1.

Table 7–1 SYS$INPUT and SYS$OUTPUT Values

Logical Name	User Mode	Equivalence Device or File
SYS$INPUT	Interactive	Terminal at which the user is logged in
	Batch job	Data lines following the invocation of the program
	Command procedure	Data lines following the invocation of the program
SYS$OUTPUT	Interactive	Terminal at which the user is logged in
	Batch job	Batch log file
	Command procedure	Terminal at which the user is logged in

Generally, use of SYS$INPUT and SYS$OUTPUT as the primary input and output devices is recommended. A user of the program can redefine SYS$INPUT and SYS$OUTPUT to redirect input and output as desired. For example, the interactive user might redefine SYS$OUTPUT as a file name to save output in a file rather than display it on the terminal.

7.2.2 Reading and Writing to Alternate Devices and External Files

Alternatively, you can design your program to read input from and write output to a file or a device other than the user's terminal. Files may be useful for writing large amounts of data, for writing data that the user might want to save, and for writing data that can be reused as input. If you use files or devices other than SYS$INPUT and SYS$OUTPUT, you should provide the names of the files or devices (best form is to use logical names) and any conventions for their use. You can specify such information by having the program write it to the terminal, by creating a help file, or by providing user documentation.

7.3 Working with Simple User I/O

Usually, you can request information from or provide information to the user with little regard for formatting. For such simple I/O, use LIB$GET_INPUT and LIB$PUT_OUTPUT or the I/O statements for your programming language.

To provide complex screen displays for input or output, use the screen management facility described in Section 7.4.

7.3.1 Default Devices for Simple I/O

The LIB$GET_INPUT and LIB$PUT_OUTPUT routines read from SYS$INPUT and write to SYS$OUTPUT, respectively. The logical names SYS$INPUT and SYS$OUTPUT are implicit to the routines, because you need only call the routine to access the I/O unit (device or file) associated with SYS$INPUT and SYS$OUTPUT. You cannot use these routines to access an I/O unit other than the one associated with SYS$INPUT or SYS$OUTPUT.

7.3.2 Getting a Line of Input

A read operation transfers one record from the input unit to a variable or variables of your choice. At 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 (carriage return), which is generated by pressing the Return key.

If you are reading character data, LIB$GET_INPUT is a simple way of prompting for and reading the data. If you are reading noncharacter data, programming language I/O statements are preferable since they allow you to translate the data to a format of your choice.

For example, Fortran 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 want to truncate the input deliberately. Overflowing the input variable using LIB$GET_INPUT causes the fatal error LIB$_INPSTRTRU (defined in $LIBDEF); overflowing the input variable using language I/O statements may not cause an error but does truncate your data.

LIB$GET_INPUT places the characters read in a variable of your choice. You must define the variable type as a character. Optionally, LIB$GET_INPUT places the number of characters read in another variable of your choice. For input at a terminal, LIB$GET_INPUT optionally writes a prompt before reading the input. The prompt is suppressed automatically for an operation not taking place at a terminal.

Example 7–1 uses LIB$GET_INPUT to read a line of input.

Example 7–1 Reading a Line of Data

```
INTEGER*4     STATUS,
2             LIB$GET_INPUT
INTEGER*2     INPUT_SIZE
CHARACTER*512 INPUT
STATUS = LIB$GET_INPUT (INPUT,           ! Input value
2                       'Input value: ', ! Prompt (optional)
2                       INPUT_SIZE)      ! Input size (optional)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.3.3 Getting Several Lines of Input

The usual technique for getting a variable number of input records—either values for which you are prompting or data records from a file—is to read and process records until the 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. That is, a sequence of data lines ends at the next DCL command (a line in the procedure beginning with a dollar sign [$]) or at the end of the command procedure file.
- File—The end of an actual file has been reached.

Process the records in a loop (one record per iteration) and terminate the loop on end-of-file. LIB$GET_INPUT returns the error RMS$_EOF (defined in $RMSDEF) when end-of-file occurs.

Example 7–2 uses a Fortran READ statement in a loop to read a sequence of integers from SYS$INPUT.

Example 7–2 Reading a Varying Number of Input Records

```
! Return status and error codes
INTEGER   STATUS,
2         IOSTAT,
3         STATUS_OK,
4         IOSTAT_OK
PARAMETER (STATUS_OK = 1,
2          IO_OK = 0)
INCLUDE   '($FORDEF)'
! Data record read on each iteration
INTEGER   INPUT_NUMBER
! Accumulated data records
INTEGER   STORAGE_COUNT,
2         STORAGE_MAX
PARAMETER (STORAGE_MAX = 255)
INTEGER    STORAGE_NUMBER (STORAGE_MAX)
```

(continued on next page)

Example 7–2 (Cont.) Reading a Varying Number of Input Records

```
! 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
```

7.3.4 Writing Simple Output

You can use LIB$PUT_OUTPUT to write character data. If you are writing noncharacter data, programming language I/O statements are preferable because they allow you to translate the data to a format of your choice.

LIB$PUT_OUTPUT writes one record of output to SYS$OUTPUT. Typically, you should avoid writing records that exceed the device width. The width of a terminal is 80 or 132 characters, depending on the setting of the physical characteristics of the device. The width of a line printer is 132 characters. If your output record exceeds the width of the device, the excess characters are either truncated or wrapped to the next line, depending on the setting of the physical characteristics.

You must define a value (a variable, constant, or expression) to be written. The value must be expressed in characters. You should specify the exact number of characters being written and not include the trailing portion of a variable.

The following example writes a character expression to SYS$OUTPUT:

```
INTEGER*4    STATUS,
2            LIB$PUT_OUTPUT
CHARACTER*40 ANSWER
INTEGER*4    ANSWER_SIZE
    .
    .
    .
STATUS = LIB$PUT_OUTPUT ('Answer: ' // ANSWER (1:ANSWER_SIZE))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4 Working with Complex User I/O

DECwindows Motif for OpenVMS (DECwindows Motif), and the SMG$ run-time library routines enable complex screen display I/O. The DECwindows Motif environment provides a consistent user interface for developing software applications. It also includes an extensive set of programming libraries and tools. The following DECwindows Motif software allows you to build a graphical user interface:

- Toolkit composed on graphical user interface objects, such as widgets and gadgets. Widgets provide advanced programming capabilities that permit you to create graphic applications easily; gadgets, similar to widgets, require less memory to create labels, buttons, and separators.
- Language to describe visual aspects of objects, such as menus, labels, and forms, and to specify changes resulting from user interaction.
- OSF/Motif Window Manager to allow you to customize the interface.

DECwindows Motif environment also makes available the LinkWorks services for creating, managing, and traversing informational links between different application-specific data. Along with the LinkWorks Manager application, LinkWorks services help organize information into a hyperinformation environment. LinkWorks Developer's Tools provide a development environment for creating, modifying, and maintaining hyperapplications.

For information about using OpenVMS DECwindows Motif, see the *Overview of DECwindows Motif for OpenVMS Documentation* and the *DECwindows Motif Guide to Application Programming*.

The SMG$ run-time library routines provide a simple, device-independent interface for managing the appearance of the terminal screen. The SMG$ routines are primarily for use with video terminals; however, they can be used with files or hardcopy terminals.

To use the screen management facility for output, do the following:

1. Create a pasteboard—A pasteboard is a logical, two-dimensional area on which you place virtual displays. Use the SMG$CREATE_PASTEBOARD routine to create a pasteboard, and associate it with a physical device. When you refer to the pasteboard, SMG performs the necessary I/O operation to the device.
2. Create a virtual display—A virtual display is a logical, two-dimensional area in which you place the information to be displayed. Use the SMG$CREATE_VIRTUAL_DISPLAY routine to create a virtual display.
3. Paste virtual displays to the pasteboard—To make a virtual display visible, map (or paste) it to the pasteboard using the SMG$PASTE_VIRTUAL_DISPLAY routine. You can reference a virtual display regardless of whether that display is currently pasted to a pasteboard.
4. Create a viewport for a virtual display—A viewport is a rectangular viewing area that can be moved around on a buffer to view different pieces of the buffer. The viewport is associated with a virtual display.

Example 7–3 associates a pasteboard with the terminal, creates a virtual display the size of the terminal screen, and pastes the display to the pasteboard. When text is written to the virtual display, the text appears on the terminal screen.

Example 7–3 Associating a Pasteboard with a Terminal

```
   .
   .
   .
! Screen management control structures
INTEGER*4 PBID,   ! Pasteboard ID
2         VDID,   ! Virtual display ID
2         ROWS,   ! Rows on screen
2         COLS    ! Columns on screen
! Status variable and routines called as functions
INTEGER*4 STATUS,
2         SMG$CREATE_PASTEBOARD,
2         SMG$CREATE_VIRTUAL_DISPLAY,
2         SMG$PASTE_VIRTUAL_DISPLAY
! Set up SYS$OUTPUT for screen management
! and get the number of rows and columns on the screen
STATUS = SMG$CREATE_PASTEBOARD (PBID,    ! Return value
2                               'SYS$OUTPUT',
2                               ROWS,    ! Return value
2                               COLUMNS) ! Return value
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create virtual display that pastes to the full screen size
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (ROWS,
2                                    COLUMNS,
2                                    VDID) ! Return value
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

(continued on next page)

Example 7–3 (Cont.) Associating a Pasteboard with a Terminal

```
! Paste virtual display to pasteboard
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (VDID,
2                                   PBID,
2                                   1, ! Starting at row 1 and
2                                   1) ! column 1 of the screen
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
```

To use the SMG$ routines for input, you associate a virtual keyboard with a physical device or file using the SMG$CREATE_VIRTUAL_KEYBOARD routine. The SMG$ input routines can be used alone or with the output routines. This section assumes that you are using the input routines with the output routines. Section 7.5 describes how to use the input routines without the output routines.

The screen management facility keeps an internal representation of the screen contents; therefore, it is important that you do not mix SMG$ routines with other forms of terminal I/O. The following subsections contain guidelines for using most of the SMG$ routines; for more details, see the *OpenVMS RTL Screen Management (SMG$) Manual*.

7.4.1 Pasteboards

Use the SMG$CREATE_PASTEBOARD routine to create a pasteboard and associate it with a physical device. SMG$CREATE_PASTEBOARD returns a unique pasteboard identification number; use that number to refer to the pasteboard in subsequent calls to SMG$ routines. After associating a pasteboard with a device, your program references only the pasteboard. The screen management facility performs all necessary operations between the pasteboard and the physical device. Example 7–4 creates a pasteboard.

Example 7–4 Creating a Pasteboard

```
STATUS = SMG$CREATE_PASTEBOARD (PBID, ROWS, COLUMNS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.1.1 Erasing a Pasteboard

When you create a pasteboard, the screen management facility clears the screen by default. To clear the screen yourself, invoke the SMG$ERASE_PASTEBOARD routine. Any virtual displays associated with the pasteboard are removed from the screen, but their contents in memory are not affected. The following example erases the screen:

```
STATUS = SMG$ERASE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.1.2 Deleting a Pasteboard

Invoking the SMG$DELETE_PASTEBOARD routine deletes a pasteboard, making the screen unavailable for further pasting. The optional second argument of the SMG$DELETE_PASTEBOARD routine allows you to indicate whether the routine clears the screen (the default) or leaves it as is. The following example deletes a pasteboard and clears the screen:

```
STATUS = SMG$DELETE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

By default, the screen is erased when you create a pasteboard. Generally, you should erase the screen at the end of a session.

7.4.1.3 Setting Screen Dimensions and Background Color

The SMG$CHANGE_PBD_CHARACTERISTICS routine sets the dimensions of the screen and its background color. You can also use this routine to retrieve dimensions and background color. To get more detailed information about the physical device, use the SMG$GET_PASTEBOARD_ATTRIBUTES routine. Example 7–5 changes the screen width to 132 and the background to white, then restores the original width and background before exiting.

Example 7–5 Modifying Screen Dimensions and Background Color

```
   .
   .
   .
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))
```

7.4.2 Virtual Displays

You write to virtual displays, which are logically configured as rectangles, by using the SMG$ routines. The dimensions of a virtual display are designated vertically as rows and horizontally as columns. A position in a virtual display is designated by naming a row and a column. Row and column numbers begin at 1.

7.4.2.1 Creating a Virtual Display

Use the SMG$CREATE_VIRTUAL_DISPLAY routine to create a virtual display. SMG$CREATE_VIRTUAL_DISPLAY returns a unique virtual display identification number; use that number to refer to the virtual display.

Optionally, you can use the fifth argument of SMG$CREATE_VIRTUAL_DISPLAY to specify one or more of the following video attributes: blinking, bolding, reversing background, and underlining. All characters written to that display will have the specified attribute unless you indicate otherwise when writing text to the display. The following example makes everything written to the display HEADER_VDID appear bold by default:

```
INCLUDE '($SMGDEF)'
   .
   .
   .
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (1,   ! Rows
2                                    80,  ! Columns
2                                    HEADER_VDID,,
2                                    SMG$M_BOLD)
```

You can border a virtual display by specifying the fourth argument when you invoke SMG$CREATE_VIRTUAL_DISPLAY. You can label the border with the routine SMG$LABEL_BORDER. If you use a border, you must leave room for it: a border requires two rows and two columns more than the size of the display. The following example places a labeled border around the STATS_VDID display. As pasted, the border occupies rows 2 and 13 and columns 1 and 57.

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (10,  ! Rows
2                                    55,  ! Columns
2                                    STATS_VDID,
2                                    SMG$M_BORDER)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$LABEL_BORDER (STATS_VDID,
2                          'statistics')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                   PBID,
2                                   3,  ! Row
2                                   2)  ! Column
```

7.4.2.2 Pasting Virtual Displays

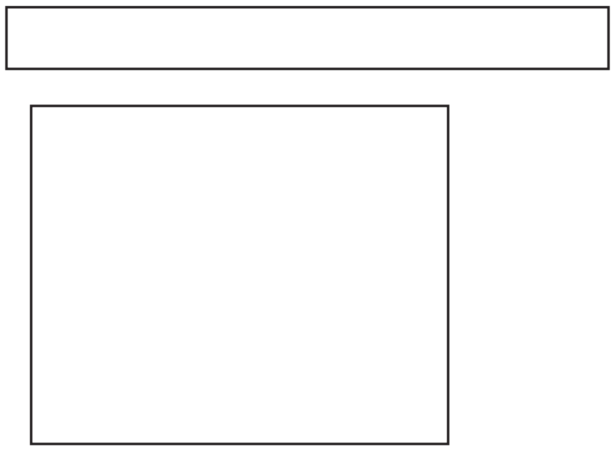
To make a virtual display visible, paste it to a pasteboard using the SMG$PASTE_VIRTUAL_DISPLAY routine. You position the virtual display by specifying which row and column of the pasteboard should contain the upper left corner of the display. Example 7–6 defines two virtual displays and pastes them to one pasteboard.

Example 7–6 Defining and Pasting a Virtual Display

```
   .
   .
   .
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                                    78,           ! Columns
2                                    HEADER_VDID,  ! Name
2                                    SMG$M_BORDER) ! Border
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (HEADER_VDID,
2                                   PBID,
2                                   2,             ! Row
2                                   2)             ! Column
IF (.NOT. STATUS) CALL LIB$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))
   .
   .
   .
```

Figure 7–1 shows the screen that results from Example 7–6.

Figure 7–1 Defining and Pasting Virtual Displays



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 Figure 7–2, displays 1 and 2 are partially occluded.) When you unpaste a virtual display that occludes another virtual display, the occluded part of the display underneath becomes visible again.

You can find out whether a display is occluded by using the SMG$CHECK_FOR_OCCLUSION routine. 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
```

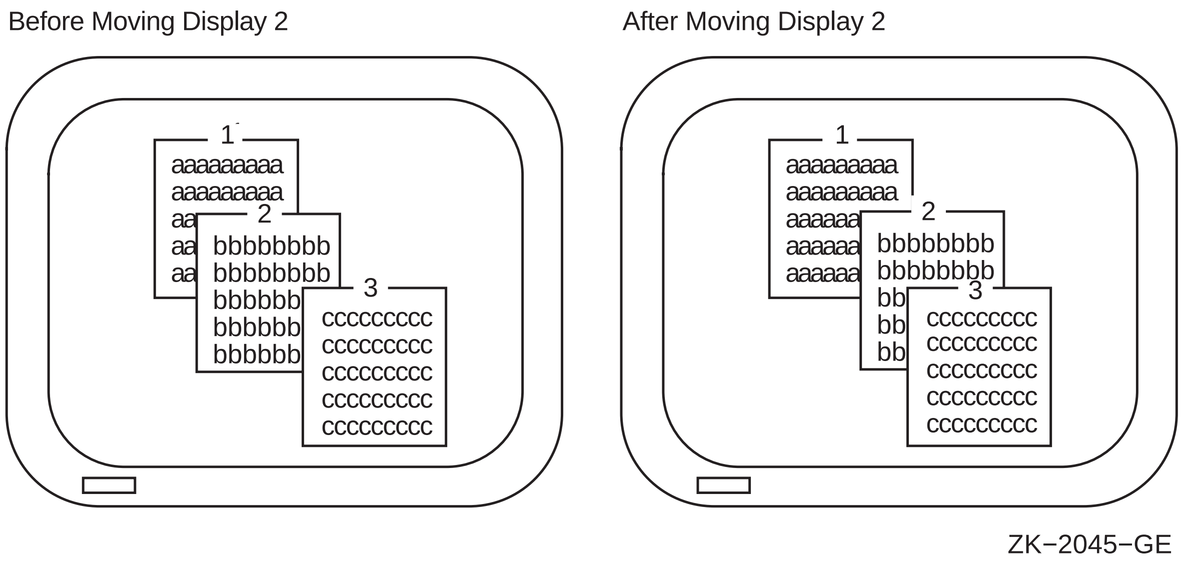
7.4.2.3 Rearranging Virtual Displays

Pasted displays can be rearranged by moving or repasting.

- Moving—To move a display, use the SMG$MOVE_VIRTUAL_DISPLAY routine. The following example moves display 2. Figure 7–2 shows the screen before and after the statement executes.

```
STATUS = SMG$MOVE_VIRTUAL_DISPLAY (VDID,
2                                  PBID,
2                                  5,
2                                  10)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

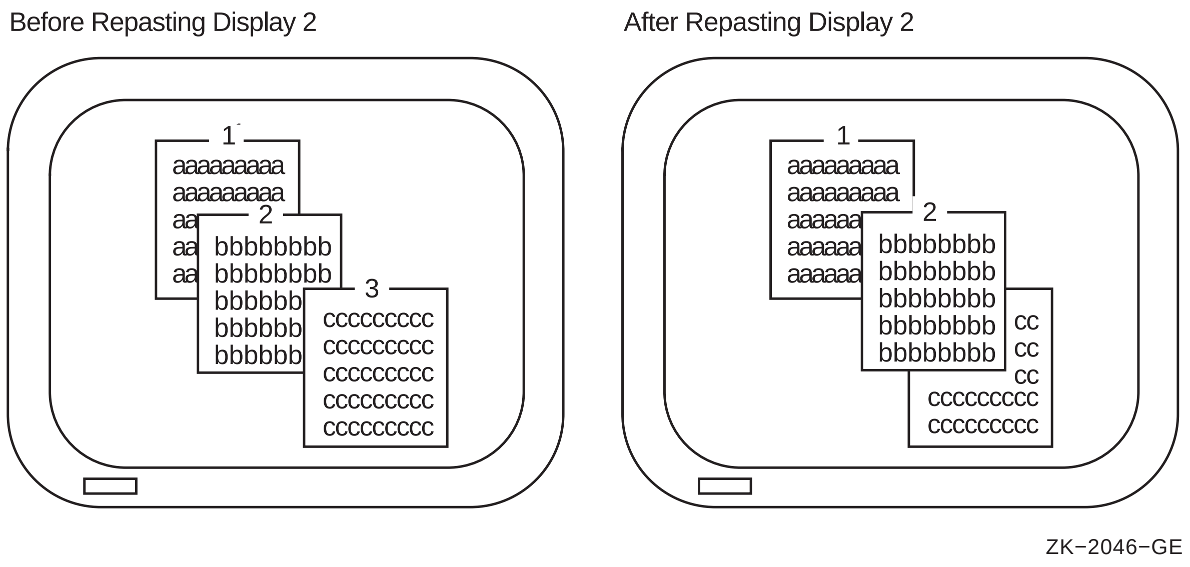
Figure 7–2 Moving a Virtual Display



- Repasting—To repaste a display, use the SMG$REPASTE_VIRTUAL_DISPLAY routine. The following example repastes display 2. Figure 7–3 shows the screen before and after the statement executes.

```
STATUS = SMG$REPASTE_VIRTUAL_DISPLAY (VDID,
2                                     PBID,
2                                     4,
2                                     4)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
```

Figure 7–3 Repasting a Virtual Display



You can obtain the pasting order of the virtual displays using SMG$LIST_ PASTING_ORDER. This routine returns the identifiers of all the virtual displays pasted to a specified pasteboard.

7.4.2.4 Removing Virtual Displays

You can remove a virtual display from a pasteboard in a number of different ways:

- Erase a virtual display—Invoking SMG$UNPASTE_VIRTUAL_DISPLAY erases a virtual display from the screen but retains its contents in memory. The following example erases the statistics display:

```
STATUS = SMG$UNPASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                    PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

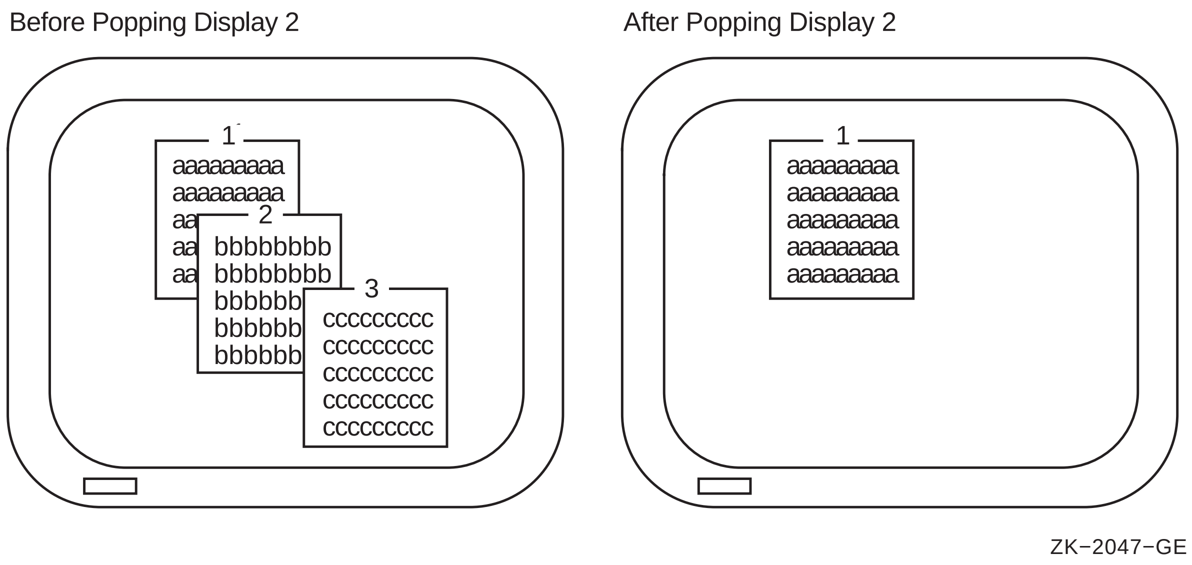
- Delete a virtual display—Invoking SMG$DELETE_VIRTUAL_DISPLAY removes a virtual display from the screen and removes its contents from memory. The following example deletes the statistics display:

```
STATUS = SMG$DELETE_VIRTUAL_DISPLAY (STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- Delete several virtual displays—Invoking SMG$POP_VIRTUAL_DISPLAY removes a specified virtual display and any virtual displays pasted after that display from the screen and removes the contents of those displays from memory. The following example “pops” display 2. Figure 7–4 shows the screen before and after popping. (Note that display 3 is deleted because it was pasted after display 2, and not because it is occluding display 2.)

```
STATUS = SMG$POP_VIRTUAL_DISPLAY (STATS_VDID,
2                                PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Figure 7–4 Popping a Virtual Display



7.4.2.5 Modifying a Virtual Display

The screen management facility provides several routines for modifying the characteristics of an existing virtual display:

- SMG$CHANGE_VIRTUAL_DISPLAY—Changes the size, video attributes, or border of a display
- SMG$CHANGE_RENDITION—Changes the video attributes of a portion of a display
- SMG$MOVE_TEXT—Moves text from one virtual display to another

The following example uses SMG$CHANGE_VIRTUAL_DISPLAY to change the size of the WHOOPS display to five rows and seven columns and to turn off all of the display's default video attributes. If you decrease the size of a display that is on the screen, any characters in the excess area are removed from the screen.

```
STATUS = SMG$CHANGE_VIRTUAL_DISPLAY (WHOOPS_VDID,
2                                    5,  ! Rows
2                                    7,, ! Columns
2                                    0)  ! Video attributes off
```

The following example uses SMG$CHANGE_RENDITION to direct attention to the first 20 columns of the statistics display by setting the reverse video attribute to the complement of the display's default setting for that attribute:

```
STATUS = SMG$CHANGE_RENDITION (STATS_VDID,
2                              1,            ! Row
2                              1,            ! Column
2                              10,           ! Number of rows
2                              20,           ! Number of columns
2                              ,             ! Video-set argument
2                              SMG$M_REVERSE) ! Video-comp argument
2
```

SMG$CHANGE_RENDITION uses three sets of video attributes to determine the attributes to apply to the specified portion of the display: (1) the display's default video attributes, (2) the attributes specified by the **rendition-set** argument of SMG$CHANGE_RENDITION, and (3) the attributes specified by the **rendition-complement** argument of SMG$CHANGE_RENDITION. Table 7–2 shows the result of each possible combination.

Table 7–2 Setting Video Attributes

rendition-set	rendition-complement	Result
off	off	Uses display default
on	off	Sets attribute
off	on	Uses the complement of display default
on	on	Clears attribute

In the preceding example, the reverse video attribute is set in the **rendition-complement** argument but not in the **rendition-set** argument, thus specifying that SMG$CHANGE_RENDITION use the complement of the display's default setting to ensure that the selected portion of the display is easily seen.

Note that the resulting attributes are based on the display's default attributes, not its current attributes. If you use SMG$ routines that explicitly set video attributes, the current attributes of the display may not match its default attributes.

7.4.2.6 Using Spawned Subprocesses

You can create a spawned subprocess directly with an SMG$ routine to allow execution of a DCL command from an application. Only one spawned subprocess is allowed per virtual display. Use the following routines to work with subprocesses:

- SMG$CREATE_SUBPROCESS—Creates a DCL spawned subprocess and associates it with a virtual display.
- SMG$EXECUTE_COMMAND—Allows execution of a specified command in the created spawned subprocess by using mailboxes. Some restrictions apply to specifying the following commands:
  - SPAWN, GOTO, or LOGOUT cannot be used and will result in unpredictable results.
  - Single-character commands such as Ctrl/C have no effect. You can signal an end-of-file (that is, press Ctrl/Z) command by setting the **flags** argument.
  - A dollar sign ($) must be specified as the first character of any DCL command.
- SMG$DELETE_SUBPROCESS—Deletes the subprocess created by SMG$CREATE_SUBPROCESS.

7.4.3 Viewports

Viewports allow you to view different pieces of a virtual display by moving a rectangular area around on the virtual display. Only one viewport is allowed for each virtual display. Once you have associated a viewport with a virtual display, the only part of the virtual display that is viewable is contained in the viewport.

The SMG$ routines for working with viewports include the following:

- SMG$CREATE_VIEWPORT—Creates a viewport and associates it with a virtual display. You must create the virtual display first. To view the viewport, you must paste the virtual display first with SMG$PASTE_ VIRTUAL_DISPLAY.
- SMG$SCROLL_VIEWPORT—Scrolls the viewport within the virtual display. If you try to move the viewport outside of the virtual display, the viewport is truncated to stay within the virtual display. This routine allows you to specify the direction and extent of the scroll.
- SMG$CHANGE_VIEWPORT—Moves the viewport to a new starting location and changes the size of the viewport.
- SMG$DELETE_VIEWPORT—Deletes the viewport and dissociates it from the virtual display. The viewport is automatically unpasted. The virtual display associated with the viewport remains intact. You can unpaste a viewport without deleting it by using SMG$UNPASTE_VIRTUAL_DISPLAY.

7.4.4 Writing Text to Virtual Display

The SMG$ output routines allow you to write text to displays and to delete or modify the existing text of a display. Remember that changes to a virtual display are visible only if the virtual display is pasted to a pasteboard.

7.4.4.1 Positioning the Cursor

Each virtual display has its own logical cursor position. You can control the position of the cursor in a virtual display with the following routines:

- SMG$HOME_CURSOR—Moves the cursor to a corner of the virtual display. The default corner is the upper left corner, that is, row 1, column 1 of the display.
- SMG$SET_CURSOR_ABS—Moves the cursor to a specified row and column.
- SMG$SET_CURSOR_REL—Moves the cursor to offsets from the current cursor position. A negative value means up (rows) or left (columns). A value of 0 means no movement.

In addition, many routines permit you to specify a starting location other than the current cursor position for the operation.

The SMG$RETURN_CURSOR_POS routine returns the row and column of the current cursor position within a virtual display. You do not have to write special code to track the cursor position.

Typically, the physical cursor is at the logical cursor position of the most recently written-to display. If necessary, you can use the SMG$SET_PHYSICAL_CURSOR routine to set the physical cursor location.

7.4.4.2 Writing Data Character by Character

If you are writing character by character (see Section 7.4.4.3 for line-oriented output), you can use three routines:

- SMG$DRAW_CHAR—Puts one line-drawing character on the screen at a specified position. It does not change the cursor position.
- SMG$PUT_CHARS—Puts one or more characters on the screen at a specified position, with the option of one video attribute.
- SMG$PUT_CHARS_MULTI—Puts several characters on the screen at a specified position, with multiple video attributes.

These routines are simple and precise. They place exactly the specified characters on the screen, starting at a specified position in a virtual display. Anything currently in the positions written-to is overwritten; no other positions on the screen are affected. Convert numeric data to character data with language I/O statements before invoking SMG$PUT_CHARS.

The following example converts an integer to a character string and places it at a designated position in a virtual display:

```
CHARACTER*4 HOUSE_NO_STRING
INTEGER*4   HOUSE_NO,
2           LINE_NO,
2           STATS_VDID
   .
   .
   .
WRITE (UNIT=HOUSE_NO_STRING,
2      FMT='(I4)') HOUSE_NO
STATUS = SMG$PUT_CHARS (STATS_VDID,
2                       HOUSE_NO_STRING,
2                       LINE_NO,  ! Row
2                       1)        ! Column
```

Note that the converted integer is right-justified from column 4 because the format specification is I4 and the full character string is written. To left-justify a converted number, you must locate the first nonblank character and write a substring starting with that character and ending with the last character.

Inserting and Overwriting Text

To insert characters rather than overwrite the current contents of the screen, use the routine SMG$INSERT_CHARS. Existing characters at the location written to are shifted to the right. Characters pushed out of the display are truncated; no wrapping occurs and the cursor remains at the end of the last character inserted.

Specifying Double-Size Characters

In addition to the preceding routines, you can use SMG$PUT_CHARS_WIDE to write characters to the screen in double width or SMG$PUT_CHARS_HIGHWIDE to write characters to the screen in double height and double width. When you use these routines, you must allot two spaces for each double-width character on the line and two lines for each line of double-height characters. You cannot mix single-and double-size characters on a line.

All the character routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the characters being written. SMG$PUT_CHARS_MULTI allows you to specify more than one video attribute at a time. The explanation of the SMG$CHANGE_RENDITION routine in Section 7.4.2.5 discusses how to use **rendition-set** and **rendition-complement** arguments.

7.4.4.3 Writing Data Line by Line

The SMG$PUT_LINE and SMG$PUT_LINE_MULTI routines write lines to virtual displays one line after another. If the display area is full, it is scrolled. You do not have to keep track of which line you are on. All routines permit you to scroll forward (up); SMG$PUT_LINE and SMG$PUT_LINE_MULTI permit you to scroll backward (down) as well. SMG$PUT_LINE permits spacing other than single spacing.

Example 7–7 writes lines from a buffer to a display area. The output is scrolled forward if the buffer contains more lines than the display area.

Example 7–7 Scrolling Forward Through a Display

```
INTEGER*4     BUFF_COUNT,
2             BUFF_SIZE (4096)
CHARACTER*512 BUFF (4096)
   .
   .
   .
DO I = 1, BUFF_COUNT
  STATUS = SMG$PUT_LINE (VDID,
2                        BUFF (I) (1:BUFF_SIZE (I)))
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
```

Example 7–8 scrolls the output backward.

Example 7–8 Scrolling Backward Through a Display

```
DO I = BUFF_COUNT, 1, -1
  STATUS = SMG$PUT_LINE (VDID,
2                        BUFF (I) (1:BUFF_SIZE (I)),
2                        SMG$M_DOWN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
```

Cursor Movement and Scrolling

To maintain precise control over cursor movement and scrolling, you can write with SMG$PUT_CHARS and scroll explicitly with SMG$SCROLL_DISPLAY_AREA. SMG$PUT_CHARS leaves the cursor after the last character written and does not force scrolling; SMG$SCROLL_DISPLAY_AREA scrolls the current contents of the display forward, backward, or sideways without writing to the display. To restrict the scrolling region to a portion of the display area, use the SMG$SET_DISPLAY_SCROLL_REGION routine.

Inserting and Overwriting Text

To insert text rather than overwrite the current contents of the screen, use the SMG$INSERT_LINE routine. Existing lines are shifted up or down to open space for the new text. If the text is longer than a single line, you can specify whether or not you want the excess characters to be truncated or wrapped.

Using Double-Width Characters

In addition, you can use SMG$PUT_LINE_WIDE to write a line of text to the screen using double-width characters. You must allot two spaces for each double-width character on the line. You cannot mix single- and double-width characters on a line.

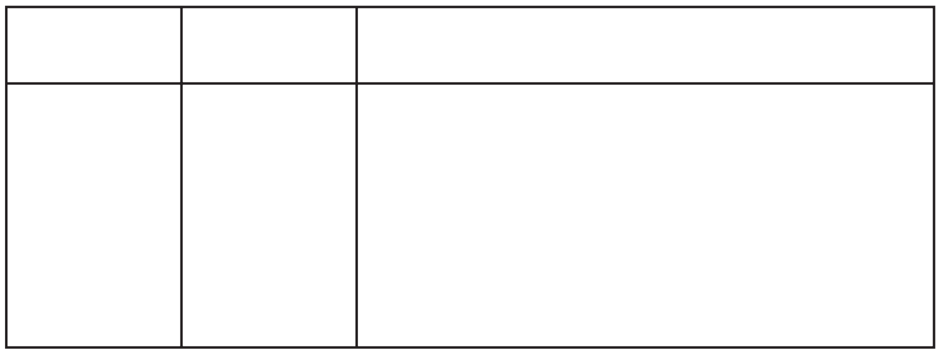
Specifying Special Video Attributes

All line routines provide **rendition-set** and **rendition-complement** arguments, which allow you to specify special video attributes for the text being written. SMG$PUT_LINE_MULTI allows you to specify more than one video attribute for the text. The explanation of the SMG$CHANGE_RENDITION routine in Section 7.4.2.5 discusses how to use the **rendition-set** and **rendition-complement** arguments.

7.4.4.4 Drawing Lines

The routine SMG$DRAW_LINE draws solid lines on the screen. Appropriate corner and crossing marks are drawn when lines join or intersect. The routine SMG$DRAW_CHARACTER draws a single character. You can also use the routine SMG$DRAW_RECTANGLE to draw a solid rectangle. Suppose that you want to draw an object such as that shown in Figure 7–5 in the statistics display area (an area of 10 rows by 55 columns).

Figure 7–5 Statistics Display



Example 7–9 shows how you can create a statistics display using SMG$DRAW_LINE and SMG$DRAW_RECTANGLE.

Example 7–9 Creating a Statistics Display

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (10,
2                                    55,
2                                    STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Draw rectangle with upper left corner at row 1 column 1
! and lower right corner at row 10 column 55
STATUS =SMG$DRAW_RECTANGLE (STATS_VDID,
2                           1, 1,
2                           10, 55)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Draw vertical lines at columns 11, 21, and 31
DO I = 11, 31, 10
  STATUS = SMG$DRAW_LINE (STATS_VDID,
2                         1, I,
2                         10, I)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
END DO
```

(continued on next page)

Example 7–9 (Cont.) Creating a Statistics Display

```
! Draw horizontal line at row 3
STATUS = SMG$DRAW_LINE (STATS_VDID,
2                       3, 1,
2                       3, 55)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (STATS_VDID,
2                                   PBID,
2                                   3,
2                                   2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.4.5 Deleting Text

The following routines erase specified characters, leaving the rest of the screen intact:

- SMG$ERASE_CHARS—Erases specified characters on one line.
- SMG$ERASE_LINE—Erases the characters on one line starting from a specified position.
- SMG$ERASE_DISPLAY—Erases specified characters on one or more lines.
- SMG$ERASE_COLUMN—Erases a column from the specified row to the end of the column from the virtual display.

The following routines perform delete operations. In a delete operation, characters following the deleted characters are shifted into the empty space.

- SMG$DELETE_CHARS—Deletes specified characters on one line. Any characters to the right of the deleted characters are shifted left.
- SMG$DELETE_LINE—Deletes one or more full lines. Any remaining lines in the display are scrolled up to fill the empty space.

The following example erases the remaining characters on the line whose line number is specified by LINE_NO, starting at the column specified by COLUMN_NO:

```
STATUS = SMG$ERASE_LINE (STATS_VDID,
2                        LINE_NO,
2                        COLUMN_NO)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.4.5 Using Menus

You can use SMG$ routines to set up menus to read user input. The type of menus you can create include the following:

- Block menu—Selections are in matrix format. This is the type of menu often used.
- Vertical menu—Each selection is on its own line.
- Horizontal menu—All selections are on one line.

Menus are associated with a virtual display, and only one menu can be used for each virtual display.

The menu routines include the following:

- SMG$CREATE_MENU—Creates a menu associated with a virtual display. This routine allows you to specify the type of menu, the position in which the menu is displayed, the format of the menu (single or double spaced), and video attributes.
- SMG$SELECT_FROM_MENU—Sets up menu selection capability. You can specify a default menu selection (which is shown in reverse video), whether online help is available, a maximum time limit for making a menu selection, a key indicating read termination, whether to send the text of the menu item selected to a string, and a video attribute.
- SMG$DELETE_MENU—Discontinues access to the menu and erases it.

When you are using menus, no other output should be sent to the menu area; otherwise, unpredictable results may occur.

The default SMG$SELECT_FROM_MENU allows specific operations, such as use of the arrow keys to move up and down the menu selections, keys to make a menu selection, ability to select more than one item at a time, ability to reselect an item already selected, and the key sequence to invoke online help. By using the **flags** argument to modify this operation, you have the option of disallowing reselection of a menu item and of allowing any key pressed to select an item.

7.4.6 Reading Data

You can read text from a virtual display (SMG$READ_FROM_DISPLAY) or from a virtual keyboard (SMG$READ_STRING, SMG$READ_COMPOSED_LINE, or SMG$READ_KEYSTROKE). The three routines for virtual keyboard input are known as the SMG$ input routines. SMG$READ_FROM_DISPLAY is not a true input routine because it reads text from the virtual display rather than from a user.

The SMG$ input routines can be used alone or with the SMG$ output routines. This section assumes that you are using the input routines with the output routines. Section 7.5 describes how to use the input routines without the output routines.

When you use the SMG$ input routines with the SMG$ output routines, always specify the optional **vdid** argument of the input routine, which specifies the virtual display in which the input is to occur. The specified virtual display must be pasted to the device associated with the virtual keyboard that is specified as the first argument of the input routine. The display must be pasted in column 1, cannot be occluded, and cannot have any other display to its right; input begins at the current cursor position, but the cursor must be in column 1.

7.4.6.1 Reading from a Display

You can read the contents of the display using the routine SMG$READ_FROM_DISPLAY. By default, the read operation reads all of the characters from the current cursor position to the end of that line. The **row** argument of SMG$READ_FROM_DISPLAY allows you to choose the starting point of the read operation, that is, the contents of the specified row to the rightmost column in that row.

If the **terminator-string** argument is specified, SMG$READ_FROM_DISPLAY searches backward from the current cursor position and reads the line beginning at the first **terminator** encountered (or at the beginning of the line). A terminator is a character string. You must calculate the length of the character string read operation yourself.

The following example reads the current contents of the first line in the STATS_VDID display:

```
CHARACTER*4 STRING
INTEGER*4   SIZE
   .
   .
   .
STATUS = SMG$HOME_CURSOR (STATS_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SMG$READ_FROM_DISPLAY (STATS_VDID,
2                               STRING)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
SIZE = 55
DO WHILE ((STRING (SIZE:SIZE) .EQ. ' ') .AND.
2         (SIZE .GT. 1))
  SIZE = SIZE - 1
END DO
```

7.4.6.2 Reading from a Virtual Keyboard

The SMG$CREATE_VIRTUAL_KEYBOARD routine establishes a device for input operations; the default device is the user's terminal. The routine SMG$READ_STRING reads characters typed on the screen until the user types a terminator or until the maximum size (which defaults to 512 characters) is exceeded. (The terminator is usually a carriage return; see the routine description in the *OpenVMS RTL Screen Management (SMG$) Manual* for a complete list of terminators.) The current cursor location for the display determines where the read operation begins.

The operating system's terminal driver processes carriage returns differently than the SMG$ routines. Therefore, in order to scroll input accurately, you must keep track of your vertical position in the display area. Explicitly set the cursor position and scroll the display. If a read operation takes place on a row other than the last row of the display, advance the cursor to the beginning of the next row before the next operation. If a read operation takes place on the last row of the display, scroll the display with SMG$SCROLL_DISPLAY_AREA and then set the cursor to the beginning of the row. Modify the read operation with TRM$M_TM_NOTRMECHO to ensure that no extraneous scrolling occurs.

Example 7–10 reads input until Ctrl/Z is pressed.

Example 7–10 Reading Data from a Virtual Keyboard

```
   .
   .
   .
! Read first record
STATUS = SMG$HOME_CURSOR (VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (KBID,
2                         TEXT,
2                         'Prompt: ',
2                         4,
2                         TRM$M_TM_TRMNOECHO,,,
2                         TEXT_SIZE,,
2                         VDID)
```

(continued on next page)

Example 7–10 (Cont.) Reading Data from a Virtual Keyboard

```
! Read remaining records until CTRL/Z
DO WHILE (STATUS .NE. SMG$_EOF)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Process record
   .
   .
   .
  ! Set up screen for next read
  ! Display area contains four rows
  STATUS = SMG$RETURN_CURSOR_POS (VDID, ROW, COL)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  IF (ROW .EQ. 4) THEN
    STATUS = SMG$SCROLL_DISPLAY_AREA (VDID)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = SMG$SET_CURSOR_ABS (VDID, 4, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ELSE
    STATUS = SMG$SET_CURSOR_ABS (VDID,, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
    STATUS = SMG$SET_CURSOR_REL (VDID, 1)
    IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  END IF
  ! Read next record
  STATUS = SMG$READ_STRING (KBID,
2                           TEXT,
2                           'Prompt: ',
2                           4,
2                           TRM$M_TM_TRMNOECHO,,,
2                           TEXT_SIZE,,
2                           VDID)
END DO
```

Note

Because you are controlling the scrolling, SMG$PUT_LINE and SMG$PUT_LINE_MULTI might not scroll as expected. When scrolling a mix of input and output, you can prevent problems by using SMG$PUT_CHARS.

7.4.6.3 Reading from the Keypad

To read from the keypad in keypad mode (that is, pressing a keypad character to perform some special action rather than entering data), modify the read operation with TRM$M_TM_ESCAPE and TRM$M_TM_NOECHO. Examine the terminator to determine which key was pressed.

Example 7–11 moves the cursor on the screen in response to the user's pressing the keys surrounding the keypad 5 key. The keypad 8 key moves the cursor north (up); the keypad 9 key moves the cursor northeast; the keypad 6 key moves the cursor east (right); and so on. The SMG$SET_CURSOR_REL routine is called, instead of being invoked as a function, because you do not want to abort the program on an error. (The error attempts to move the cursor out of the display area and, if this error occurs, you do not want the cursor to move.) The read operation is also modified with TRM$M_TM_PURGE to prevent the user from getting ahead of the cursor.

See Section 7.4.6.1 for the guidelines for reading from the display.

Example 7–11 Reading Data from the Keypad

```
   .
   .
   .
INTEGER STATUS,
2       PBID,
2       ROWS,
2       COLUMNS,
2       VDID,     ! Virtual display ID
2       KID,      ! Keyboard ID
2       SMG$CREATE_PASTEBOARD,
2       SMG$CREATE_VIRTUAL_DISPLAY,
2       SMG$CREATE_VIRTUAL_KEYBOARD,
2       SMG$PASTE_VIRTUAL_DISPLAY,
2       SMG$HOME_CURSOR,
2       SMG$SET_CURSOR_REL,
2       SMG$READ_STRING,
2       SMG$ERASE_PASTEBOARD,
2       SMG$PUT_CHARS,
2       SMG$READ_FROM_DISPLAY
CHARACTER*31 INPUT_STRING,
2            MENU_STRING
INTEGER*2    TERMINATOR
INTEGER*4    MODIFIERS
INCLUDE '($SMGDEF)'
INCLUDE '($TRMDEF)'
! Set up screen and keyboard
STATUS = SMG$CREATE_PASTEBOARD (PBID,
2                               'SYS$OUTPUT',
2                               ROWS,
2                               COLUMNS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (ROWS,
2                                    COLUMNS,
2                                    VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PUT_CHARS (VDID,
2                       '__ MENU CHOICE ONE',
2                       10,30)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PUT_CHARS (VDID,
2                       '__ MENU CHOICE TWO',
2                       15,30)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (KID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (VDID,
2                                   PBID,
2                                   1,
2                                   1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Put cursor in NW corner
STATUS = SMG$HOME_CURSOR (VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

(continued on next page)

Example 7–11 (Cont.) Reading Data from the Keypad

```
! Read character from keyboard
MODIFIERS = TRM$M_TM_ESCAPE .OR.
2           TRM$M_TM_NOECHO .OR.
2           TRM$M_TM_PURGE
STATUS = SMG$READ_STRING (KID,
2                         INPUT_STRING,
2                         ,
2                         6,
2                         MODIFIERS,
2                         ,
2                         ,
2                         ,
2                         TERMINATOR)
DO WHILE ((STATUS) .AND.
2         (TERMINATOR .NE. SMG$K_TRM_CR))
  ! Check status of last read
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! North
  IF (TERMINATOR .EQ. SMG$K_TRM_KP8) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, 0)
  ! Northeast
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP9) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, 1)
  ! Northwest
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP7) THEN
    CALL SMG$SET_CURSOR_REL (VDID, -1, -1)
  ! South
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP2) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, 0)
  ! Southeast
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP3) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, 1)
  ! Southwest
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP1) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 1, -1)
  ! East
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP6) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 0, 1)
  ! West
  ELSE IF (TERMINATOR .EQ. SMG$K_TRM_KP4) THEN
    CALL SMG$SET_CURSOR_REL (VDID, 0, -1)
  END IF
  ! Read another character
  STATUS = SMG$READ_STRING (KID,
2                           INPUT_STRING,
2                           ,
2                           6,
2                           MODIFIERS,
2                           ,
2                           ,
2                           ,
2                           TERMINATOR)
END DO
```

(continued on next page)

Example 7–11 (Cont.) Reading Data from the Keypad

```
! Read menu entry and process
!
STATUS = SMG$READ_FROM_DISPLAY (VDID,
2                               MENU_STRING)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
! Clear screen
STATUS = SMG$ERASE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

7.4.6.4 Reading Composed Input

The SMG$CREATE_KEY_TABLE routine creates a table that equates keys to character strings. When you read input using the routine SMG$READ_COMPOSED_LINE and the user presses a defined key, the corresponding character string in the table is substituted for the key. The SMG$ADD_KEY_DEF routine can be used to load the table. Composed input also permits the following:

- If states—You can define the same key to mean different things in different states. You can define a key to cause a change in state. The change in state can be temporary (until after the next defined key is pressed) or permanent (until a key that changes states is pressed).
- Input termination—You can define the key to cause termination of the input transmission (as if the Return key were pressed after the character string). If the key is not defined to cause termination of the input, the user must press a terminator or another key that does cause termination.

Example 7–12 defines keypad keys 1 through 9 and permits the user to change state temporarily by pressing the PF1 key. Pressing the keypad 1 key is equivalent to typing 1000 and pressing the Return key. Pressing PF1 key and then the keypad 1 key is equivalent to typing 10000 and pressing the Return key.

Example 7–12 Redefining Keys

```
INTEGER*4 TABLEID
   .
   .
   .
! Create table for key definitions
STATUS = SMG$CREATE_KEY_TABLE (TABLEID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Load table
! If user presses PF1, the state changes to BYTEN
! The BYTEN state is in effect only for the very next key
STATUS = SMG$ADD_KEY_DEF (TABLEID,
2                         'PF1',
2                         ,,,'BYTEN')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

(continued on next page)

Example 7–12 (Cont.) Redefining Keys

```
! 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))
```

(continued on next page)

Example 7–12 (Cont.) Redefining Keys

```
! 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 SMG$DELETE_KEY_DEF routine to delete a key definition; use the SMG$GET_KEY_DEF routine to examine a key definition. You can also load key definition tables with the SMG$DEFINE_KEY and SMG$LOAD_KEY_DEFS routines; use the DCL command DEFINE/KEY to specify input to these routines.

To use keypad keys 0 through 9, the keypad must be in application mode. For details, see SMG$SET_KEYPAD_MODE in the *OpenVMS RTL Screen Management (SMG$) Manual*.

7.4.7 Controlling Screen Updates

If your program needs to make a number of changes to a virtual display, you can use SMG$ routines to make all of the changes before updating the display. The SMG$BEGIN_DISPLAY_UPDATE routine causes output operations to a pasted display to be reflected only in the display's buffers. The SMG$END_DISPLAY_UPDATE routine 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 sent to the pasteboard immediately. The counter mechanism allows a subroutine to request and turn off batching without disturbing the batching state of the calling program.

A second set of routines, SMG$BEGIN_PASTEBOARD_UPDATE and SMG$END_PASTEBOARD_UPDATE, allow you to buffer output to a pasteboard in a similar manner.

7.4.8 Maintaining Modularity

When using the SMG$ routines, you must take care not to corrupt the mapping between the screen appearance and the internal representation of the screen. Therefore, observe the following guidelines:

- Mixing SMG I/O and other forms of I/O

  In general, do not use any other form of terminal I/O while the terminal is active as a pasteboard. If you do use I/O other than SMG I/O (for example, if you invoke a subprogram that may perform non-SMG terminal I/O), first invoke the SMG$SAVE_PHYSICAL_SCREEN routine and when the non-SMG I/O completes, invoke the SMG$RESTORE_PHYSICAL_SCREEN routine, as demonstrated in the following example:

```
STATUS = SMG$SAVE_PHYSICAL_SCREEN (PBID,
2                                  SAVE_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CALL GET_EXTRA_INFO (INFO_ARRAY)
STATUS = SMG$RESTORE_PHYSICAL_SCREEN (PBID,
2                                     SAVE_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

- Sharing the pasteboard

  A routine using the terminal screen without consideration for its current contents must use the existing pasteboard ID associated with the terminal and delete any virtual displays it creates before returning control to the high-level code. This guideline also applies to the program unit that invokes a subprogram that also performs screen I/O. The safest way to clean up your virtual displays is to call the SMG$POP_VIRTUAL_DISPLAY routine and name the first virtual display you created. The following example invokes a subprogram that uses the terminal screen:

  Invoking Program Unit

```
CALL GET_EXTRA_INFO (PBID,
2                    INFO_ARRAY)
   .
   .
   .
CALL STATUS = SMG$CREATE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

  Subprogram

```
SUBROUTINE GET_EXTRA_INFO (PBID,
2                          INFO_ARRAY)
   .
   .
   .
! Start executable code
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (4,
2                                    40,
2                                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (INSTR_VDID,
2                                   PBID, 1, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
STATUS = SMG$POP_VIRTUAL_DISPLAY (INSTR_VDID,
2                                 PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

- Sharing virtual displays

  To share a virtual display created by high-level code, the low-level code must use the virtual display ID created by the high-level code; an invoking program unit must pass the virtual display ID to the subprogram. To share a virtual display created by low-level code, the high-level code must use the virtual display ID created by the low-level code; a subprogram must return the virtual display ID to the invoking program.

The following example permits a subprogram to use a virtual display created by the invoking program unit:

Invoking Program Unit

```
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (4,
2                                    40,
2                                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$PASTE_VIRTUAL_DISPLAY (INSTR_VDID,
2                                   PBID, 1, 1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CALL GET_EXTRA_INFO (PBID,
2                    INSTR_VDID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

Subprogram

```
SUBROUTINE GET_EXTRA_INFO (PBID,
2                          INSTR_VDID)
```

7.5 Performing Special Input/Output Actions

Screen management input routines and the SYS$QIO and SYS$QIOW system services allow you to perform I/O operations otherwise unavailable to high-level languages. For example, you can allow a user to interrupt normal program execution by typing a character and by providing a mechanism for reading that character. You can also control such things as echoing, time allowed for input, and whether data is read from the type-ahead buffer.

Some of the operations described in the following sections require the use of the SYS$QIO or SYS$QIOW system services. For more information about the QIO system services, see the *OpenVMS System Services Reference Manual* and Chapter 9.

Other operations, described in the following sections, can be performed by calling the SMG$ input routines. The SMG$ input routines can be used alone or with the SMG$ output routines. Section 7.4 describes how to use the input routines with the output routines. This section assumes that you are using the input routines alone. To use the SMG$ input routines, do the following:

1. Call SMG$CREATE_VIRTUAL_KEYBOARD to associate a logical keyboard with a device or file specification (SYS$INPUT by default). SMG$CREATE_VIRTUAL_KEYBOARD returns a keyboard identification number; use that number to identify the device or file to the SMG$ input routines.
2. Call an SMG$ input routine (SMG$READ_STRING or SMG$READ_COMPOSED_LINE) to read data typed at the device associated with the virtual keyboard.

When using the SMG$ input routines without the SMG$ output routines, do not specify the optional VDID argument of the input routine.

7.5.1 Using Ctrl/C and Ctrl/Y Interrupts

The QIO system services enable you to detect a Ctrl/C or Ctrl/Y interrupt at a user terminal, even if you have not issued a read to the terminal. To do so, you must take the following steps:

1. Queue an asynchronous system trap (AST)—Issue the SYS$QIO or SYS$QIOW system service with a function code of IO$_SETMODE modified by either IO$M_CTRLCAST (for Ctrl/C interrupts) or IO$M_CTRLYAST (for Ctrl/Y interrupts). For the **P1** argument, provide the

name of a subroutine to be executed when the interrupt occurs. For the **P2** argument, you can optionally identify one longword argument to pass to the AST subroutine.

2. Write an AST subroutine—Write the subroutine identified in the **P1** argument of the QIO system service and link the subroutine into your program. Your subroutine can take one longword dummy argument to be associated with the **P2** argument in the QIO system service. You must define common areas to access any other data in your program from the AST routine.

If you press 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 unless another AST has been queued). Note the following guidelines for using Ctrl/C and Ctrl/Y ASTs:

- ASTs are asynchronous—Since your AST subroutine does not know exactly where you are in your program when the interrupt occurs, you should avoid manipulating data or performing other mainline activities. In general, the AST subroutine should either notify the mainline code (for example, by setting a flag) that the interrupt occurred, or clean up and exit from the program (if that is what you want to do).
- ASTs need new channels to the terminal—If you try to access the terminal with language I/O statements using SYS$INPUT or SYS$OUTPUT, you may receive a redundant I/O error. You must establish another channel to the terminal by explicitly opening the terminal.
- Ctrl/C and Ctrl/Y ASTs are one-time ASTs—After a Ctrl/C or Ctrl/Y AST is delivered, it is dequeued. You must reissue the QIO system service if you wish to trap another interrupt.
- Many ASTs can be queued—You can queue multiple ASTs (for the same or different AST subroutines, on the same or different channels) by issuing the appropriate number of QIO system services. The system delivers the ASTs on a last-in, first-out (LIFO) basis.
- Unhandled Ctrl/Cs turn into Ctrl/Ys—If the user enters Ctrl/C and you do not have an AST queued to handle the interrupt, the system turns the Ctrl/C interrupt into a Ctrl/Y interrupt.
- DCL handles Ctrl/Y interrupts—DCL handles Ctrl/Y interrupts by returning the user to DCL command level, where the user has the option of continuing or exiting from your program. DCL takes precedence over your AST subroutine for Ctrl/Y interrupts. Your Ctrl/Y AST subroutine is executed only under the following circumstances:
  - 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 LIB$DISABLE_CTRL
  - If the user elects to continue your program after DCL interrupts it
- You can dequeue Ctrl/C and Ctrl/Y ASTs—You can dequeue all Ctrl/C or Ctrl/Y ASTs on a channel by issuing the appropriate QIO system service with a value of 0 for the **P1** argument (passed by immediate value). You can dequeue all Ctrl/C ASTs on a channel by issuing the SYS$CANCEL system

service for the appropriate channel. You can dequeue all Ctrl/Y ASTs on a channel by issuing the SYS$DASSGN system service for the appropriate channel.

- You can use SMG$ routines—You can connect to the terminal using the SMG$ routines from either AST level or mainline code. Do not attempt to connect to the terminal from AST level if you do so in your mainline code.

Example 7–13 permits the terminal user to interrupt a display to see how many lines have been typed up to that point.

Example 7–13 Using Interrupts to Perform I/O

```
!Main Program
   .
   .
   .
INTEGER STATUS
! Accumulated data records
CHARACTER*132 STORAGE (255)
INTEGER*4     STORAGE_SIZE (255),
2             STORAGE_COUNT
! QIOW and QIO structures
INTEGER*2 INPUT_CHAN
INTEGER*4 CODE
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT
  BYTE      TRANSMIT,
2           RECEIVE,
2           CRFILL,
2           LFFILL,
2           PARITY,
2           ZERO
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Flag to notify program of CTRL/C interrupt
LOGICAL*4 CTRLC_CALLED
! AST subroutine to handle CTRL/C interrupt
EXTERNAL CTRLC_AST
! Subroutines
INTEGER SYS$ASSIGN,
2       SYS$QIOW
! Symbols used for I/O operations
INCLUDE '($IODEF)'
! Put values into array
CALL LOAD_STORAGE (STORAGE,
2                  STORAGE_SIZE,
2                  STORAGE_COUNT)
! Assign channel and set up QIOW structures
STATUS = SYS$ASSIGN ('SYS$INPUT',
2                    INPUT_CHAN,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
CODE = IO$_SETMODE .OR. IO$M_CTRLCAST
```

(continued on next page)

Example 7–13 (Cont.) Using Interrupts to Perform I/O

```
! 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
```

7.5.2 Detecting Unsolicited Input

You can detect input from the terminal even if you have not called SMG$READ_COMPOSED_LINE or SMG$READ_STRING by using SMG$ENABLE_UNSOLICITED_INPUT. This routine uses the AST mechanism to transfer control to a subprogram of your choice each time the user types at the terminal; the AST subprogram is responsible for reading any input. When the subprogram completes, control returns to the point in your mainline code where it was interrupted.

The SMG$ENABLE_UNSOLICITED_INPUT routine is not an SMG$ input routine. Before invoking SMG$ENABLE_UNSOLICITED_INPUT, you must invoke SMG$CREATE_PASTEBOARD to associate a pasteboard with the terminal and SMG$CREATE_VIRTUAL_KEYBOARD to associate a virtual keyboard with the same terminal.

SMG$ENABLE_UNSOLICITED_INPUT accepts the following arguments:

- The pasteboard identification number (use the value returned by SMG$CREATE_PASTEBOARD)
- The name of an AST subprogram
- An argument to be passed to the AST subprogram

When SMG$ENABLE_UNSOLICITED_INPUT invokes the AST subprogram, it passes two arguments to the subprogram: the pasteboard identification number and the argument that you specified. Typically, you write the AST subprogram to read the unsolicited input with SMG$READ_STRING. Since SMG$READ_STRING requires that you specify the virtual keyboard at which the input was typed, specify the virtual keyboard identification number as the second argument to pass to the AST subprogram.

Example 7–14 permits the terminal user to interrupt the display of a series of arrays, and either to 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).

Example 7–14 Receiving Unsolicited Input from a Virtual Keyboard

```
! Main Program
! The main program calls DISPLAY_ARRAY once for each array.
! DISPLAY_ARRAY displays the array in a DO loop.
! If the user enters input from the terminal, the loop is
! interrupted and the AST routine takes over.
! If the user types anything beginning with an N, the AST
! sets DO_NEXT and resumes execution -- DISPLAY_ARRAY drops
! out of the loop processing the array (because DO_NEXT is
! set -- and the main program calls DISPLAY_ARRAY for the
! next array.
! If the user types anything not beginning with an N,
! the program exits.
   .
   .
   .
INTEGER*4 STATUS,
2         VKID,  ! Virtual keyboard ID
2         PBID   ! Pasteboard ID
! Storage arrays
INTEGER*4 ARRAY1 (256),
2         ARRAY2 (256),
2         ARRAY3 (256)
! System routines
INTEGER*4 SMG$CREATE_PASTEBOARD,
2         SMG$CREATE_VIRTUAL_KEYBOARD,
2         SMG$ENABLE_UNSOLICITED_INPUT
! AST routine
EXTERNAL  AST_ROUTINE
```

(continued on next page)

Example 7–14 (Cont.) Receiving Unsolicited Input from a Virtual Keyboard

```
! Create a pasteboard
STATUS = SMG$CREATE_PASTEBOARD (PBID,        ! Pasteboard ID
2                                'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Create a keyboard for the same device
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,  ! Keyboard ID
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Enable unsolicited input
STATUS = SMG$ENABLE_UNSOLICITED_INPUT (PBID, ! Pasteboard ID
2                                      AST_ROUTINE,
2                                      VKID) ! Pass keyboard
                                             ! ID to AST
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .
! Call display subroutine once for each array
CALL DISPLAY_ARRAY (ARRAY1)
CALL DISPLAY_ARRAY (ARRAY2)
CALL DISPLAY_ARRAY (ARRAY3)

END
```

Array Display Routine

```
! Subroutine to display one array
SUBROUTINE DISPLAY_ARRAY (ARRAY)
! Dummy argument
INTEGER*4 ARRAY (256)
! Status
INTEGER*4 STATUS
! Flag for doing next array
LOGICAL*4 DO_NEXT
COMMON /DO_NEXT/ DO_NEXT
! If AST has been delivered, reset
IF (DO_NEXT) DO_NEXT = .FALSE.
! Initialize control variable
I = 1
! Display entire array unless interrupted by user
! If interrupted by user (DO_NEXT is set), drop out of loop
DO WHILE ((I .LE. 256) .AND. (.NOT. DO_NEXT))
  TYPE *, ARRAY (I)
  I = I + 1
END DO

END
```

(continued on next page)

Example 7–14 (Cont.) Receiving Unsolicited Input from a Virtual Keyboard

AST Routine

```
! Subroutine to read unsolicited input
SUBROUTINE AST_ROUTINE (PBID,
2                       VKID)
! dummy arguments
INTEGER*4 PBID,                  ! Pasteboard ID
2         VKID                   ! Keyboard ID
! Status
INTEGER*4 STATUS
! Flag for doing next array
LOGICAL*4 DO_NEXT
COMMON /DO_NEXT/ DO_NEXT
! Input string
CHARACTER*4 INPUT
! Routines
INTEGER*4 SMG$READ_STRING
! Read input
STATUS = SMG$READ_STRING (VKID,  ! Keyboard ID
2                         INPUT)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! If user types anything beginning with N, set DO_NEXT
! otherwise, exit from program
IF (INPUT (1:1) .EQ. 'N') THEN
  DO_NEXT = .TRUE.
ELSE
  CALL EXIT
END IF

END
```

7.5.3 Using the Type-Ahead Buffer

Normally, if the user types at the terminal before your application is able to read from that device, the input is saved in a special data structure maintained by the system called the type-ahead buffer. When your application is ready to read from the terminal, the input is transferred from the type-ahead buffer to your input buffer. The type-ahead buffer is preset at a size of 78 bytes. If the HOSTSYNC characteristic is on (the usual condition), input to the type-ahead buffer is stopped (the keyboard locks) when the buffer is within 8 bytes of being full. If the HOSTSYNC characteristic is off, the bell rings when the type-ahead buffer is within 8 bytes of being full; if you overflow the buffer, the excess data is lost. The TTY_ALTALARM system parameter determines the point at which either input is stopped or the bell rings.

You can clear the type-ahead buffer by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_PURGE in the **modifiers** argument. Clearing the type-ahead buffer has the effect of reading only what the user types on the terminal after the read operation is invoked. Any characters in the type-ahead buffer are lost. The following example illustrates how to purge the type-ahead buffer:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,      ! Virtual keyboard ID
2             INPUT_SIZE
```

```
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,
2                                     'SYS$INPUT') ! I/O device
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,     ! Keyboard ID
2                         INPUT,    ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_PURGE,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also clear the type-ahead buffer with a QIO read operation modified by IO$M_PURGE (defined in $IODEF). You can turn off the type-ahead buffer for further read operations with a QIO set mode operation that specifies TT$M_ NOTYPEAHD as a basic terminal characteristic.

You can examine the type-ahead buffer by issuing a QIO sense mode operation modified by IO$M_TYPEAHDCNT. The number of characters in the type-ahead buffer and the value of the first character are returned to the **P1** argument.

The size of the type-ahead buffer is determined by the TTY_TYPAHDSZ system parameter. You can specify an alternative type-ahead buffer by turning on the ALTYPEAHD terminal characteristic; the size of the alternative type-ahead buffer is determined by the TTY_ALTYPAHD system parameter.

7.5.4 Using Echo

Normally, the system writes back to the terminal any printable characters that the user types at that terminal. The system also writes highlighted words in response to certain control characters; for example, the system writes EXIT if the user enters Ctrl/Z. If the user types ahead of your read, the characters are not echoed until you read them from the type-ahead buffer.

You can turn off echoing when you invoke a read operation by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_NOECHO in the **modifiers** argument. You can turn off echoing for control characters only by modifying the read operation with TRM$M_TM_TRMNOECHO. The following example turns off all echoing for the read operation:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,       ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,        ! Keyboard ID
2                                     'SYS$INPUT') ! I/O device
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,        ! Keyboard ID
2                         INPUT,       ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_NOECHO,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also turn off echoing with a QIO read operation modified by IO$M_NOECHO (defined in $IODEF). You can turn off echoing for further read operations with a QIO set mode operation that specifies TT$M_NOECHO as a basic terminal characteristic.

7.5.5 Using Timeout

Using SMG$READ_STRING, you can restrict the user to a certain amount of time in which to respond to a read command. If your application reads data from the terminal using SMG$READ_STRING, you can modify the timeout characteristic by specifying, in the **timeout** argument, the number of seconds the user has to respond. If the user fails to type a character in the allotted time, the error condition SS$_TIMEOUT (defined in $SSDEF) is returned. The following example restricts the user to 8 seconds in which to respond to a read command:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,     ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($SSDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,      ! Keyboard ID
2                         INPUT,     ! Data read
2                         'Prompt> ',
2                         512,
2                         ,
2                         8,
2                         ,
2                         INPUT_SIZE)
IF (.NOT. STATUS) THEN
  IF (STATUS .EQ. SS$_TIMEOUT) CALL NO_RESPONSE ()
ELSE
  CALL LIB$SIGNAL (%VAL (STATUS))
END IF
```

You can cause a QIO read operation to time out after a certain number of seconds by modifying the operation with IO$M_TIMED and by specifying the number of seconds in the **P3** argument. A message broadcast to a terminal resets a timer that is set for a timed read operation (regardless of whether the operation was initiated with QIO or SMG).

Note that the timed read operations work on a character-by-character basis. To set a time limit on an input record rather than an input character, you must use the SYS$SETIMR system service. The SYS$SETIMR executes an AST routine at a specified time. The specified time is the input time limit. When the specified time is reached, the AST routine cancels any outstanding I/O on the channel that is assigned to the user's terminal.

7.5.6 Converting Lowercase to Uppercase

You can automatically convert lowercase user input to uppercase by reading from the terminal with the SMG$READ_STRING routine and by specifying TRM$M_TM_CVTLOW in the **modifiers** argument, as shown in the following example:

```
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 to uppercase with a QIO read operation modified by IO$M_CVTLOW (defined in $IODEF).

7.5.7 Performing Line Editing and Control Actions

Normally, the user can edit input as explained in the *OpenVMS I/O User's Reference Manual*. You can inhibit line editing on the read operation by reading from the terminal with SMG$READ_STRING and by specifying TRM$M_TM_NOFILTR in the **modifiers** argument. The following example shows how you can inhibit line editing:

```
INTEGER*4     SMG$CREATE_VIRTUAL_KEYBOARD,
2             SMG$READ_STRING,
2             STATUS,
2             VKID,     ! Virtual keyboard ID
2             INPUT_SIZE
CHARACTER*512 INPUT
INCLUDE       '($TRMDEF)'
STATUS = SMG$CREATE_VIRTUAL_KEYBOARD (VKID,  ! Keyboard ID
2                                     'SYS$INPUT')
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$READ_STRING (VKID,   ! Keyboard ID
2                         INPUT,  ! Data read
2                         'Prompt> ',
2                         512,
2                         TRM$M_TM_NOFILTR,
2                         ,,
2                         INPUT_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

You can also inhibit line editing with a QIO read operation modified by IO$M_NOFILTR (defined in $IODEF).

7.5.8 Using Broadcasts

You can write, or broadcast, to any interactive terminal by using the SYS$BRKTHRU system service. The following example broadcasts a message to all terminals at which users are currently logged in. Use of SYS$BRKTHRU to write to a terminal allocated to a process other than your own requires the OPER privilege.

```
INTEGER*4 STATUS,
2         SYS$BRKTHRUW
INTEGER*2 B_STATUS (4)
INCLUDE   '($BRKDEF)'
STATUS = SYS$BRKTHRUW (,
2                      'Accounting system started',,
2                      %VAL (BRK$C_ALLUSERS),
2                      B_STATUS,,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
```

7.5.8.1 Default Handling of Broadcasts

If the terminal user has taken no action to handle broadcasts, a broadcast is written to the terminal screen at the current position (after a carriage return and line feed). If a write operation is in progress, the broadcast occurs after the write ends. If a read operation is in progress, the broadcast occurs immediately; after the broadcast, any echoed user input to the aborted read operation is written to the screen (same effect as pressing Ctrl/R).

7.5.8.2 How to Create Alternate Broadcast Handlers

You can handle broadcasts to the terminal on which your program is running with SMG$SET_BROADCAST_TRAPPING. This routine uses the AST mechanism to transfer control to a subprogram of your choice each time a broadcast message is sent to the terminal; when the subprogram completes, control returns to the point in your mainline code where it was interrupted.

The SMG$SET_BROADCAST_TRAPPING routine is not an SMG$ input routine. Before invoking SMG$SET_BROADCAST_TRAPPING, you must invoke SMG$CREATE_PASTEBOARD to associate a pasteboard with the terminal. SMG$CREATE_PASTEBOARD returns a pasteboard identification number; pass that number to SMG$SET_BROADCAST_TRAPPING to identify the terminal in question. Read the contents of the broadcast with SMG$GET_BROADCAST_MESSAGE.

Example 7–15 demonstrates how you might trap a broadcast and write it at the bottom of the screen. For more information about the use of SMG$ pasteboards and virtual displays, see Section 7.4.

Example 7–15 Trapping Broadcast Messages

```
   .
   .
   .
INTEGER*4 STATUS,
2         PBID,                                ! Pasteboard ID
2         VDID,                                ! Virtual display ID
2         SMG$CREATE_PASTEBOARD,
2         SMG$SET_BROADCAST_TRAPPING
2         SMG$PASTE_VIRTUAL_DISPLAY
COMMON    /ID/ PBID,
2              VDID
INTEGER*2 B_STATUS (4)
INCLUDE   '($SMGDEF)'
INCLUDE   '($BRKDEF)'
EXTERNAL  BRKTHRU_ROUTINE
STATUS = SMG$CREATE_PASTEBOARD (PBID)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$CREATE_VIRTUAL_DISPLAY (3,          ! Height
2                                    80,         ! Width
2                                    VDID,,      ! Display ID
2                                    SMG$M_REVERSE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
STATUS = SMG$SET_BROADCAST_TRAPPING (PBID,    ! Pasteboard ID
2                                       BRKTHRU_ROUTINE) ! AST
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
   .
   .
   .

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 File Operations

This chapter describes file operations that support file input/output (I/O) and file I/O instructions of the operating system's high-level languages. This chapter contains the following sections:

Section 8.1 describes file attributes.

Section 8.2 describes strategies to access files.

Section 8.3 describes protection and access of files.

Section 8.4 describes file mapping.

Section 8.5 describes how to open and update a sequential file.

Section 8.6 describes using the Fortran user-open routines.

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.

Note

Some confusion might arise between records in a file and record variables. Where this chapter refers to a record variable, the term *record variable* is used; otherwise, *record* refers to a record in a file.

8.1 File Attributes

Before writing a program that accesses a data file, you must know the attributes of the file and the order of the data. To determine this information, see your language-specific programming manual.

File attributes (organization, record structure, and so on) determine how data is stored and accessed. Typically, the attributes are specified by keywords when you open the data file.

Ordering of the data within a file is not important mechanically. However, if you attempt to read data without knowing how it is ordered within the file, you are likely to read the wrong data; if you attempt to write data without knowing how it is ordered within the file, you are likely to corrupt existing data.

8.1.1 Specifying File Attributes

Large sets of attributes can be specified using the File Definition Language utility (FDL). All of the file attributes can be specified using OpenVMS RMS in a user-open routine (see Section 8.6). Typically, you need only programming language file specifiers. Use FDL only when language specifiers are unavailable.

Refer to the appropriate programming language reference manual for information about the use of language specifiers.

For complete information about how to use FDL, see the *OpenVMS Record Management Utilities Reference Manual*.

8.2 File Access Strategies

When determining the file attributes and order of your data file, consider how you plan to access that data. File access strategies fall into the following categories:

- Complete access

  If your program processes all or most of the data in the file and especially if many references are made to the data, you should read the entire file into memory. Put each record in its own variable or set of variables.

  If your program is larger than the amount of virtual memory available (including the additional memory you get by using memory allocation routines), you must declare fewer variables and process your file in pieces. To determine the size of your program, add the number of bytes in each program section. The DCL command LINK/MAP produces a listing that includes the length of each program section (PSECT).

- Record-by-record access

  If your program accesses records one after another, or if you cannot fit the entire file into memory, you should read one record into memory at a time.

- Discrete records access

  If your program processes only a selection of the file's records, you should read only the necessary records into memory.

- Sequential and indexed file access

  If your program demands speed and needs to conserve disk space, use an unformatted sequential file. Use indexed files to process selected sets of records or to access records directly. Use a sequential file with fixed-length records, a relative file, or an indexed file to access records directly.

8.3 File 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 File Definition Language (FDL) attributes OWNER and PROTECTION in the file.

8.3.1 Read-Only Access

By default, the user of your program must have write access to a file in order for your program to open that file. However, if you specify use of the Fortran READONLY specifier when opening the file, the user needs only read access to the file in order to open it. The READONLY specifier does not set the protection on a file. The user cannot write to a file opened with the READONLY specifier.

8.3.2 Shared Access

The Fortran specifier READONLY and the SHARED specifier allow multiple processes to open the same file simultaneously, provided that each process uses one of these specifiers when opening the file. The READONLY specifier allows the process read access to the file; the SHARED specifier allows other processes read and write access to the file. If a process opens the file without specifying READONLY or SHARED, no other process can open that file even by specifying READONLY or SHARED.

In the following Fortran segment, if the read operation indicates that the record is locked, the read operation is repeated. You should not attempt to read a locked record without providing a delay (in this example, the call to ERRSNS) to allow the other process time to complete its operation and unlock the record.

```
! Status variables and values
INTEGER STATUS,
2       IOSTAT,
2       IO_OK
PARAMETER (IO_OK = 0)
INCLUDE '($FORDEF)'
! Logical unit number
INTEGER LUN /1/
! Record variables
INTEGER LEN
CHARACTER*80 RECORD
   .
   .
   .
READ (UNIT = LUN,
2     FMT = '(Q,A)'
2     IOSTAT = IOSTAT) LEN, RECORD (1:LEN)
IF (IOSTAT .NE. IO_OK) THEN
  CALL ERRSNS (,,,,STATUS)
   IF (STATUS .EQ. FOR$_SPERECLOC) THEN
     DO WHILE (STATUS .EQ. FOR$_SPERECLOC)
     READ (UNIT = LUN,
2           FMT = '(Q,A)'
2           IOSTAT = IOSTAT) LEN, RECORD(1:LEN)
     IF (IOSTAT .NE. IO_OK) THEN
           CALL ERRSNS (,,,,STATUS)
           IF (STATUS .NE. FOR$_SPERECLOC) THEN
                CALL LIB$SIGNAL(%VAL(STATUS))
           END IF
     END IF
     END DO
ELSE
   CALL LIB$SIGNAL (%VAL(STATUS))
   END IF
END IF
   .
   .
   .
```

In Fortran, each time you access a record in a shared file, that record is automatically locked until you perform another I/O operation on the same logical unit, or until you explicitly unlock the record using the UNLOCK statement. If you plan to modify a record, you should do so before unlocking it; otherwise, you should unlock the record as soon as possible.

8.4 File Access and Mapping

To copy an entire data file from the disk to program variables and back again, either use language I/O statements to read and write the data or use the Create and Map Section (SYS$CRMPSC) system service to map the data. Oten times, mapping the file is faster than reading it. However, a mapped file usually uses more virtual memory than one that is read using language I/O statements. Using I/O statements, you have to store only the data that you have entered. Using SYS$CRMPSC, you have to initialize the database and store the entire structure in virtual memory including the parts that do not yet contain data.

8.4.1 Using SYS$CRMPSC

Mapping a file means associating each byte of the file with a byte of program storage. You access data in a mapped file by referencing the program storage; your program does not use I/O statements.

Note

Files created using OpenVMS RMS typically contain control information. Unless you are familiar with the structure of these files, do not attempt to map one. The best practice is to map only those files that have been created as the result of mapping.

To map a file, perform the following operations:

1. Place the program variables for the data in a common block. Page align the common block at link time by specifying an options file containing the following link option for VAX and Alpha systems:

   VAX

   For VAX systems, specify the following:

   ```
   PSECT_ATTR = name, PAGE  ♦
   ```

   Alpha

   For Alpha systems, specify the following:

   ```
   PSECT_ATTR = name, solitary  ♦
   ```

   The variable *name* is the name of the common block.

   Within the common block, you should specify the data in order from most complex to least complex (high to low rank), with character data last. This naturally aligns the data, thus preventing troublesome page breaks in virtual memory.

2. Open the data file using a user-open routine. The user-open routine must open the file for user I/O (as opposed to OpenVMS RMS I/O) and return the channel number on which the file is opened.
3. Map the data file to the common block.
4. Process the records using the program variables in the common block.
5. Free the memory used by the common block, forcing modified data to be written back to the disk file.

Do not initialize variables in a common block that you plan to map; the initial values will be lost when SYS$CRMPSC maps the common block.

8.4.1.1 Mapping a File

The format for SYS$CRMPSC is as follows:

```
SYS$CRMPSC ([inadr],[retadr],[acmode],[flags],[gsdnam],[ident],[relpag],
            [chan], [pagcnt],[vbn],[prot],[pfc])
```

For a complete description of the SYS$CRMPSC system service, see the *OpenVMS System Services Reference Manual*.

Starting and Ending Addresses of the Mapped Section

VAX

On VAX systems, specify the location of the first variable in the common block as the value of the first array element of the array passed by the **inadr** argument. Specify the location of the last variable in the common block as the value of the second array element. ♦

Alpha

On Alpha systems, specify the location of the first variable in the common block as the value of the first array element of the array passed by the **inadr** argument; the second array element must be the address of the last variable in the common block, which is derived by performing a logical OR with the value of the size of a memory page minus 1. The size of the memory page can be retrieved by a call to the SYS$GETSYI system service. ♦

If the first variable in the common block is an array or string, the first variable in the common block is the first element of that array or string. If the last variable in the common block is an array or string, the last variable in the common block is the last element in that array or string.

Returning the Location of the Mapped Section

VAX

SYS$CRMPSC returns the location of the first and last elements mapped in the **retadr** argument. The value returned as the starting virtual address should be the same as the starting address passed to the **inadr** argument. The value returned as the ending virtual address should be equal to or slightly more than (within 512 bytes, or 1 block) the value of the ending virtual address passed to the **inadr** argument. ♦

Alpha

SYS$CRMPSC returns the location of the first and last elements mapped in the **retadr** argument. The value returned as the starting virtual address should be the same as the starting address passed to the **inadr** argument. The value returned as the ending virtual address should be equal to or slightly less than (within a single page size) the value of the ending virtual address passed to the **inadr** argument. ♦

If the first element is in error, you probably forgot to page-align the common block containing the mapped data.

If the second element is in error, you were probably creating a new data file and forgot to specify the size of the file in your program (see Section 8.4.1.3).

Using Private Sections

Specify SEC$M_WRT for the **flags** to indicate that the section is writable. If the file is new, also specify SEC$M_DZRO to indicate that the section should be initialized to zero.

Obtaining the Channel Number

You must use a user-open routine to get the channel number (see Section 8.4.1.2). Pass the channel number to the **chan** argument.

VAX

Example 8–1 maps a data file consisting of one longword and three real arrays to the INC_DATA common block. The options file INCOME.OPT page-aligns the INC_DATA common block.

If SYS$CRMPSC returns a status of SS$_IVSECFLG and you have correctly specified the flags in the mask argument, check to see if you are passing a channel number of 0.

Example 8–1 Mapping a Data File to the Common Block on a VAX System

```
!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)
```

(continued on next page)

Example 8–1 (Cont.) Mapping a Data File to the Common Block on a VAX System

```
IF (EXIST) THEN
  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))

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 ♦
```

Alpha

Example 8–2 shows the code for performing the same functions as Example 8–1 but in an Alpha system's environment.

Example 8–2 Mapping a Data File to the Common Block on an Alpha System

```
!INCOME.OPT

PSECT_ATTR = INC_DATA, SOLITARY, SHR, WRT
```

INCOME.FOR

```
! Declare variables to hold statistics
REAL PERSONS_HOUSE (2048),
2    ADULTS_HOUSE (2048),
2    INCOME_HOUSE (2048)
INTEGER TOTAL_HOUSES, STATUS
! 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)
```

(continued on next page)

Example 8–2 (Cont.) Mapping a Data File to the Common Block on an Alpha System

```
IF (EXIST) THEN
  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$M_DZRO
  SEC_LEN =
!  (address of last - address of first + size of last + 511)/512
2  ( (%LOC(TOTAL_HOUSES) - %LOC(PERSONS_HOUSE(1)) + 4 + 511)/512 )
  OPEN (UNIT=STATS_LUN,
2       FILE='STATS.SAV',
2       STATUS='NEW',
2       INITIALSIZE = SEC_LEN,
2       USEROPEN = UFO_CREATE)
END IF
! Free logical unit number and map section
CLOSE (STATS_LUN)
! ********
! MAP DATA
! ********
STATUS = LIB$GETSYI(SYI$_PAGE_SIZE, PAGE_MAX,,,,)
IF (.NOT. STATUS) CALL LIB$STOP (%VAL (STATUS))
! Specify first and last address of section
ADDR(1) = %LOC(PERSONS_HOUSE(1))
! Section will always be smaller than page_max bytes
ADDR(2) = ADDR(1) + PAGE_MAX -1
! 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))

2  CALL LIB$SIGNAL (%VAL (%LOC(INCOME_BADMAP)))
   .
   .
   .
                     ! Reference data using the
                     ! data structures listed
                     ! in the common block
   .
   .
   .
! Close and update STATS.SAV
STATUS = SYS$DELTVA (RET_ADDR,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$DASSGN (%VAL(CHAN))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

END ♦
```

8.4.1.2 Using the User-Open Routine

When you open a file for mapping in Fortran, for example, you must specify a user-open routine (Section 8.6 discusses user-open routines) to perform the following operations:

1. Set the user-file open bit (FAB$V_UFO) in the file access block (FAB) options mask.
2. Open the file using SYS$OPEN for an existing file or SYS$CREATE for a new file. (Do not invoke SYS$CONNECT if you have set the user-file open bit.)
3. Return the channel number to the program unit that started the OPEN operation. The channel number is in the additional status longword of the FAB (FAB$L_STV) and must be returned in a common block.
4. Return the status of the open operation (SYS$OPEN or SYS$CREATE) as the value of the user-open routine.

After setting the user-file open bit in the FAB options mask, you cannot use language I/O statements to access data in that file. Therefore, you should free the logical unit number associated with the file. The file is still open. You access the file with the channel number.

Example 8–3 shows a user-open routine invoked by the sample program in Section 8.4.1.1 if the file STATS.SAV exists. (If STATS.SAV does not exist, the user-open routine must invoke SYS$CREATE rather than SYS$OPEN.)

Example 8–3 Using a User-Open Routine

```
!UFO_OPEN.FOR

INTEGER FUNCTION UFO_OPEN (FAB,
2                          RAB,
2                          LUN)

! Include Open VMS 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
```

8.4.1.3 Initializing a Mapped Database

The first time you map a file you must perform the following operations in addition to those listed at the beginning of Section 8.4.1:

1. Specify the size of the file—SYS$CRMPSC maps data based on the size of the file. Therefore, when creating a file that is to be mapped, you must specify in your program a file large enough to contain all of the expected data. Figure the size of your database as follows:
   - Find the size of the common block (in bytes)—Subtract the location of the first variable in the common block from the location of the last variable in the common block and then add the size of the last element.
   - Find the number of blocks in the common block—Add 511 to the size and divide the result by 512 (512 bytes = 1 block).
2. Initialize the file when you map it—The blocks allocated to a file might not be initialized and therefore contain random data. When you first map the file, you should initialize the mapped area to zeros by setting the SEC$V_DZRO bit in the mask argument of SYS$CRMPSC.

The user-open routine for creating a file is the same as the user-open routine for opening a file except that SYS$OPEN is replaced by SYS$CREATE.

8.4.1.4 Saving a Mapped File

To close a data file that was 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 Delete Virtual Address Space (SYS$DELTVA) system service to delete the virtual memory used by a mapped file. Use the Deassign I/O Channel (SYS$DASSGN) system service to deassign the I/O channel assigned to a file.

The program segment shown in Example 8–4 closes a mapped file, automatically writing any modifications back to the disk. To ensure that the proper locations are deleted, pass SYS$DELTVA the addresses returned to your program by SYS$CRMPSC rather than the addresses you passed to SYS$CRMPSC. If you want to save modifications made to the mapped section without closing the file, use the Update Section File on Disk (SYS$UPDSEC) system service. To ensure that the proper locations are updated, pass SYS$UPDSEC the addresses returned to your program by SYS$CRMPSC rather than the addresses you passed to SYS$CRMPSC. Typically, you want to wait until the update operation completes before continuing program execution. Therefore, use the **efn** argument of SYS$UPDSEC to specify an event flag to be set when the update is complete, and wait for the system service to complete before continuing. For a complete description of the SYS$DELTVA, SYS$DASSGN, and SYS$UPDSEC system services, see the *OpenVMS System Services Reference Manual*.

Example 8–4 Closing a Mapped File

```
! Section address
INTEGER*4 ADDR(2),
2         RET_ADDR(2)
! Event flag
INTEGER*4 FLAG
! Status block
STRUCTURE /IO_BLOCK/
  INTEGER*2 IOSTAT,
2           HARDWARE
  INTEGER*4 BAD_PAGE
END STRUCTURE
RECORD /IO_BLOCK/ IOSTATUS
   .
   .
   .
! Get an event flag
STATUS = LIB$GET_EF (FLAG)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Update the section
STATUS = SYS$UPDSEC (RET_ADDR,
2                    ,,,
2                    %VAL(FLAG)
2                    ,
2                    IOSTATUS,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
! Wait for section to be updated
STATUS = SYS$SYNCH (%VAL(FLAG),
2                   IOSTATUS)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
   .
   .
   .
```

8.5 Opening and Updating a Sequential File

This section provides an example, written in DEC Fortran, of how to open and update a sequential file on a VAX system. A sequential file consists of records arranged one after the other in the order in which they are written to the file. Records can only be added to the end of the file. Typically, sequential files are accessed sequentially.

Creating a Sequential File

To create a sequential file, use the OPEN statement and specify the following keywords and keyword values:

- STATUS =′NEW′
- ACCESS = ′SEQUENTIAL′
- ORGANIZATION = ′SEQUENTIAL′

The file structure keyword ORGANIZATION also accepts the value ′INDEXED′ or ′RELATIVE′.

Example 8–5 creates a sequential file of fixed-length records.

Example 8–5 Creating a Sequential File of Fixed-Length Records

```
   .
   .
   .
INTEGER STATUS,
2       LUN,
2       LIB$GET_INPUT,
2       LIB$GET_LUN,
2       STR$UPCASE
INTEGER*2     FN_SIZE,
2             REC_SIZE
CHARACTER*256 FILENAME
CHARACTER*80  RECORD
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get free unit number
STATUS = LIB$GET_LUN (LUN)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the file
OPEN (UNIT = LUN,
2     FILE = FILENAME (1:FN_SIZE),
2     ORGANIZATION = 'SEQUENTIAL',
2     ACCESS = 'SEQUENTIAL',
2     RECORDTYPE = 'FIXED',
2     FORM = 'UNFORMATTED',
2     RECL = 20,
2     STATUS = 'NEW')
! Get the record input
STATUS = LIB$GET_INPUT (RECORD,
2                       'Input: ',
2                       REC_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
DO WHILE (REC_SIZE .NE. 0)

  ! Convert to uppercase
  STATUS = STR$UPCASE (RECORD,RECORD)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  WRITE  (UNIT=LUN) RECORD(1:REC_SIZE)
  ! Get more record input
  STATUS = LIB$GET_INPUT (RECORD,
2                         'Input: ',
2                         REC_SIZE)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END DO

END
```

Updating a Sequential File

To update a sequential file, read each record from the file, update it, and write it to a new sequential file. Updated records cannot be written back as replacement records for the same sequential file from which they were read.

Example 8–6 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; because the new update file was opened after the old file, the new file has a higher version number.

Example 8–6 Updating a Sequential File

```
   .
   .
   .
INTEGER STATUS,
2       LUN1,
2       LUN2,
2       IOSTAT
INTEGER*2  FN_SIZE
CHARACTER*256 FILENAME
CHARACTER*80 RECORD
CHARACTER*80 NEW_RECORD
INCLUDE '($FORDEF)'
INTEGER*4 LIB$GET_INPUT,
2         LIB$GET_LUN,
2         STR$UPCASE
! Get file name
STATUS = LIB$GET_INPUT (FILENAME,
2                       'File name: ',
2                       FN_SIZE)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Get free unit number
STATUS = LIB$GET_LUN (LUN1)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the old file
OPEN (UNIT=LUN1,
2     FILE=FILENAME (1:FN_SIZE),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=20,
2     STATUS='OLD')
! Get free unit number
STATUS = LIB$GET_LUN (LUN2)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Open the new file
OPEN (UNIT=LUN2,
2     FILE=FILENAME (1:FN_SIZE),
2     ORGANIZATION='SEQUENTIAL',
2     ACCESS='SEQUENTIAL',
2     RECORDTYPE='FIXED',
2     FORM='UNFORMATTED',
2     RECL=20,
2     STATUS='NEW')
! 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
```

(continued on next page)

Example 8–6 (Cont.) Updating a Sequential File

```
  ! Get record update
  STATUS = LIB$GET_INPUT (NEW_RECORD,
2                         'Update: ')
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
  ! Convert to uppercase
  STATUS = STR$UPCASE (NEW_RECORD,
2                      NEW_RECORD)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

  ! Write unchanged record or updated record
  IF (NEW_RECORD .EQ. ' ' ) THEN
    WRITE (UNIT=LUN2) RECORD
  ELSE
    WRITE (UNIT=LUN2) NEW_RECORD
  END IF

  ! Read the next record
  READ (UNIT=LUN1,
2       IOSTAT=IOSTAT) RECORD
  IF (IOSTAT .NE. IOSTAT_OK) THEN
    CALL ERRSNS (,,,,STATUS)
    IF (STATUS .NE. FOR$_ENDDURREA) THEN
      CALL LIB$SIGNAL (%VAL(STATUS))
    END IF
  END IF
END DO

END
```

8.6 User-Open Routines

A user-open routine in Fortran, for example, gives you direct access to the file access block (FAB) and record access block (RAB) (the OpenVMS RMS structures that define file characteristics). Use a user-open routine to specify file characteristics that are otherwise unavailable from your programming language.

When you specify a user-open routine, you open the file rather than allow the program to open the file for you. Before passing the FAB and RAB to your user-open routine, any default file characteristics and characteristics that can be specified by keywords in the programming language are set. Your user-open routine should not set or modify such file characteristics because the language might not be aware that you have set the characteristics and might not perform as expected.

8.6.1 Opening a File

Section 8.4.1.2 provides guidelines on opening a file with a user-open routine. This section provides an example of a Fortran user-open routine.

8.6.1.1 Specifying USEROPEN

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. Because 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)
   .
   .
   .
```

8.6.1.2 Writing the User-Open Routine

Write a user-open routine as an INTEGER function that accepts three dummy arguments:

- FAB address—Declare this argument as a RECORD variable. Use the record structure FABDEF defined in the $FABDEF module of SYS$LIBRARY:FORSYSDEF.TLB.
- RAB address—Declare this argument as a RECORD variable. Use the record structure RABDEF defined in the $RABDEF module of SYS$LIBRARY:FORSYSDEF.TLB.
- Logical unit number—Declare this argument as an INTEGER.

A user-open routine must perform at least the following operations. In addition, before opening the file, a user-open routine usually adjusts one or more fields in the FAB or the RAB or in both.

- Opens the file—To open the file, invoke the SYS$OPEN system service if the file already exists, or the SYS$CREATE system service if the file is being created.
- Connects the file—Invoke the SYS$CONNECT system service to establish a record stream for I/O.
- Returns the status—To return the status, equate the return status of the SYS$OPEN or SYS$CREATE system service to the function value of the user-open routine.

The following user-open routine opens an existing file. The file to be opened is specified in the OPEN statement of the invoking program unit.

UFO_OPEN.FOR

```
INTEGER FUNCTION UFO_OPEN (FAB,
2                          RAB,
2                          LUN)

! Include Open VMS RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN
```

```
! Declare status variable
INTEGER STATUS
! Declare system routines
INTEGER SYS$CREATE,
2       SYS$OPEN,
2       SYS$CONNECT
! Optional FAB and/or RAB modifications
   .
   .
   .
! Open file
STATUS = SYS$OPEN (FAB)
IF (STATUS)
2  STATUS = SYS$CONNECT (RAB)

! Return status of $OPEN or $CONNECT
UFO_OPEN = STATUS

END
```

8.6.1.3 Setting FAB and RAB Fields

Each field in the FAB and RAB is identified by a symbolic name, such as FAB$L_FOP. Where separate bits in a field represent different attributes, each bit offset is identified by a similar symbolic name, such as FAB$V_CTG. The first three letters identify the structure containing the field. The letter following the dollar sign indicates either the length of the field (B for byte, W for word, or L for longword) or that the name is a bit offset (V for bit) rather than a field. The letters following the underscore identify the attribute associated with the field or bit. The symbol FAB$L_FOP identifies the FAB options field, which is a longword in length; the symbol FAB$V_CTG identifies the contiguity bit within the options field.

The STRUCTURE definitions for the FAB and RAB are in the $FABDEF and $RABDEF modules of the library SYS$LIBRARY:FORSYSDEF.TLB. To use these definitions, do the following:

1. Include the modules in your program unit.
2. Declare RECORD variables for the FAB and the RAB.
3. Reference the various fields of the FAB and RAB using the symbolic name of the field.

The following user-open routine specifies that the blocks allocated for the file must be contiguous. To specify contiguity, you clear the best-try-contiguous bit (FAB$V_CBT) of the FAB$L_FOP field and set the contiguous bit (FAB$V_CTG) of the same field.

UFO_CONTIG.FOR

```
INTEGER FUNCTION UFO_CONTIG (FAB,
2                            RAB,
2                            LUN)

! Include Open VMS RMS definitions
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
! Declare dummy arguments
RECORD /FABDEF/ FAB
RECORD /RABDEF/ RAB
INTEGER LUN
! Declare status variable
INTEGER STATUS
```

```
! Declare system procedures
INTEGER SYS$CREATE,
2       SYS$CONNECT
! Clear contiguous-best-try bit and
! set contiguous bit in FAB options
FAB.FAB$L_FOP = IBCLR (FAB.FAB$L_FOP, FAB$V_CBT)
FAB.FAB$L_FOP = IBSET (FAB.FAB$L_FOP, FAB$V_CTG)
! Open file
STATUS = SYS$CREATE (FAB)
IF (STATUS) STATUS = SYS$CONNECT (RAB)

! Return status of open or connect
UFO_CONTIG = STATUS

END
```

9 System Service Input/Output Operations

This chapter describes how to use system services to perform input and output operations. It contains the following sections:

Section 9.1 describes the QIO operation.

Section 9.2 describes the use of quotas, privileges, and protection.

Section 9.3 describes device addressing modes.

Section 9.4 describes I/O function encoding.

Section 9.5 describes how to assign channels.

Section 9.6 describes how to queue I/O requests.

Section 9.7 describes how to synchronize I/O completions.

Section 9.8 describes the routine to use to wait for completion of an asynchronous event.

Section 9.9 describes executing I/O services synchronously or asynchronously.

Section 9.10 describes the completion status of an I/O operation.

Section 9.11 describes how to deassign I/O channels.

Section 9.12 presents a progam example of a complete input and output operation.

Section 9.13 describes how to cancel I/O requests.

Section 9.14 describes how to use logical names and physical device names for I/O operations.

Section 9.15 describes how to use device name defaults.

Section 9.16 describes how to obtain information about physical devices.

Section 9.17 describes device allocation.

Section 9.18 describes how to mount, dismount, and initialize disk and tape volumes.

Section 9.19 describes format output strings.

Section 9.20 describes how to use mailboxes for I/O operations.

Section 9.21 provides a program example of using I/O system services.

Examples are provided to show you how to use the I/O services for simple functions, such as terminal input and output operations. If you plan to write device-dependent I/O routines, see the *OpenVMS I/O User's Reference Manual*.

VAX

If you want to write your own device driver or connect to a device interrupt vector, see the *OpenVMS VAX Device Support Reference Manual*. ♦

Besides using I/O system services, you can use OpenVMS Record Management Services (RMS). OpenVMS RMS provides a set of routines for general-purpose, device-independent functions such as data storage, retrieval, and modification.

Unlike RMS services, I/O system services permit you to use the I/O resources of the operating system directly in a device-dependent manner. I/O services also provide some specialized functions not available in OpenVMS RMS. Using I/O services requires more programming knowledge than using OpenVMS RMS, but can result in more efficient input/output operations.

9.1 Overview of OpenVMS QIO Operations

The OpenVMS operating system provides QIO operations that perform three basic I/O functions: read, write, and set mode. The read function transfers data from a device to a user-specified buffer. The write function transfers data in the opposite direction—from a user-specified buffer to the device. For example, in a read QIO function to a terminal device, a user-specified buffer is filled with characters received from the terminal. In a write QIO function to the terminal, the data in a user-specified buffer is transferred to the terminal where it is displayed.

The set mode QIO function is used to control or describe the characteristics and operation of a device. For example, a set mode QIO function to a line printer can specify either uppercase or lowercase character format. Not all QIO functions are applicable to all types of devices. The line printer, for example, cannot perform a read QIO function.

9.2 Quotas, Privileges, and Protection

To preserve the integrity of the operating system, the I/O operations are performed under the constraints of quotas, privileges, and protection.

Quotas limit the number and type of I/O operations that a process can perform concurrently and the total size of outstanding transfers. They ensure that all users have an equitable share of system resources and usage.

Privileges are granted to a user to allow the performance of certain I/O-related operations, for example, creating a mailbox and performing logical I/O to a file-structured device. Restrictions on user privileges protect the integrity and performance of both the operating system and the services provided to other users.

Protection controls access to files and devices. Device protection is provided in much the same way as file protection: shareable and nonshareable devices are protected by protection masks.

The Set Resource Wait Mode (SYS$SETRWM) system service allows a process to select either of two modes when an attempt to exceed a quota occurs. In the enabled (default) mode, the process waits until the required resource is available before continuing. In the disabled mode, the process is notified immediately by a system service status return that an attempt to exceed a quota has occurred. Waiting for resources is transparent to the process when resource wait mode is enabled; the process takes no explicit action when a wait is necessary.

The different types of I/O-related quotas, privilege, and protection are described in the following sections.

9.2.1 Buffered I/O Quota

The buffered I/O limit quota (BIOLM) specifies the maximum number of concurrent buffered I/O operations that can be active in a process. In a buffered I/O operation, the user's data is buffered in system dynamic memory. The driver deals with the system buffer and not the user buffer. Buffered I/O is used for terminal, line printer, card reader, network, mailbox, and console medium transfers and file system operations. For a buffered I/O operation, the system does not have to lock the user's buffer in memory.

The system manager, or the person who creates the process, establishes the buffered I/O quota value in the user authorization file. If you use the Set Resource Wait Mode (SYS$SETRWM) system service to enable resource wait mode for the process, the process enters resource wait mode if it attempts to exceed its direct I/O quota.

9.2.2 Buffered I/O Byte Count Quota

The buffered I/O byte count quota (BYTELM) specifies the maximum amount of buffer space that can be consumed from system dynamic memory for buffering I/O requests. All buffered I/O requests require system dynamic memory in which the actual I/O operation takes place.

The system manager, or the person who creates the process, establishes the buffered I/O byte count quota in the user authorization file. If you use the SYS$SETRWM system service to enable resource wait mode for the process, the process enters resource wait mode if it attempts to exceed its direct I/O quota.

9.2.3 Direct I/O Quota

The direct I/O limit quota (DIOLM) specifies the maximum number of concurrent direct (unbuffered) I/O operations that a process can have active. In a direct I/O operation, data is moved directly to or from the user buffer. Direct I/O is used for disk, magnetic tape, most direct memory access (DMA) real-time devices, and nonnetwork transfers, such as DMC11/DMR11 write transfers. For direct I/O, the user's buffer must be locked in memory during the transfer.

The system manager, or the person who creates the process, establishes the direct I/O quota value in the user authorization file. If you use the SYS$SETRWM system service to enable resource wait mode for the process, the process enters resource wait mode if it attempts to exceed its direct I/O quota.

9.2.4 AST Quota

The AST quota specifies the maximum number of outstanding asynchronous system traps that a process can have. The system manager, or the person who creates the process, establishes the quota value in the user authorization file. There is never an implied wait for that resource.

9.2.5 Physical I/O Privilege

Physical I/O privilege (PHY_IO) allows a process to perform physical I/O operations on a device. Physical I/O privilege also allows a process to perform logical I/O operations on a device.

9.2.6 Logical I/O Privilege

Logical I/O privilege (LOG_IO) allows a process to perform logical I/O operations on a device. A process can also perform physical operations on a device if the process has logical I/O privilege, the volume is mounted foreign, and the volume protection mask allows access to the device. (A foreign volume is one volume that contains no standard file structure understood by any of the operating system software.) See Section 9.3.2 for a further information about logical I/O privilege.

9.2.7 Mount Privilege

Mount privilege (MOUNT) allows a process to use the IO$_MOUNT function to perform mount operations on disk and magnetic tape devices. The IO$_MOUNT function is used in ancillary control processs (ACP) interface operations.

9.2.8 Volume Protection

Volume protection protects the integrity of mailboxes and both foreign and Files–11 On-Disk Structure Level 2 structured volumes. Volume protection for a foreign volume is established when the volume is mounted. Volume protection for a Files–11 structured volume is established when the volume is initialized. (If the process mounting the volume has the override volume protection privilege, VOLPRO, protection can be overridden when the volume is mounted.)

The SYS$CREMBX system service protection mask argument establishes mailbox protection.

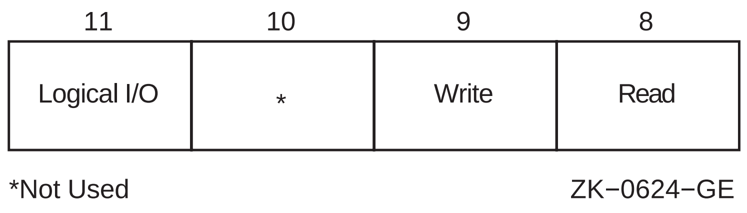
Set Protection QIO requests allow you to set volume protection on a mailbox. You must either be the owner of the mailbox or have the BYPASS privilege.

Protection for structured volumes and mailboxes is provided by a volume protection mask that contains four 4-bit fields. These fields correspond to the four classes of user permitted to access the volume. (User classes are based on the volume owner’s UIC.)

The 4-bit fields are interpreted differently for volumes that are mounted as structured (that is, volumes serviced by an ACP), volumes that are mounted as foreign, and mailboxes (both temporary and permanent).

Figure 9–1 shows the 4-bit protection fields for mailboxes. Usually, volume protection is meaningful only for read and write operations.

Figure 9–1 Mailbox Protection Fields



9.2.9 Device Protection

Device protection protects the allocation of nonshareable devices, such as terminals and card readers.

Protection is provided by a device protection mask similar to that of volume protection. The difference is that only the bit corresponding to read access is checked, and that bit determines whether the process can allocate or assign a channel to the device.

You establish device protection with the DCL command SET PROTECTION /DEVICE. This command sets both the protection mask and the device owner UIC.

9.2.10 System Privilege

System UIC privilege (SYSPRV) allows a process to be eligible for the volume or device protection specified for the system protection class, even if the process does not have a UIC in one of the system groups.

9.2.11 Bypass Privilege

Bypass privilege (BYPASS) allows a process to bypass volume and device protection completely.

9.3 Physical, Logical, and Virtual I/O

I/O data transfers can occur in any one of three device addressing modes: physical, logical, or virtual. Any process with device access allowed by the volume protection mask can perform logical I/O on a device that is mounted foreign; physical I/O requires privileges. Virtual I/O does not require privileges; however, intervention by an ACP to control user access might be necessary if the device is under ACP control. (ACP functions are described in the *OpenVMS I/O User's Reference Manual*.)

9.3.1 Physical I/O Operations

In physical I/O operations, data is read from and written to the actual, physically addressable units accepted by the hardware (for example, sectors on a disk or binary characters on a terminal in the PASSALL mode). This mode allows direct access to all device-level I/O operations.

Physical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO).
- The issuing process has all of the following characteristics:
  - The issuing process has logical I/O privilege (LOG_IO).
  - The device is mounted foreign.
  - The volume protection mask allows physical access to the device.

If neither of these conditions is met, the physical I/O operation is rejected by the SYS$QIO system service, which returns a condition value of SS$_NOPRIV (no privilege). Figure 9–2 illustrates the physical I/O access checks in greater detail.

The inhibit error-logging function modifier (IO$M_INHERLOG) can be specified for all physical I/O functions. The IO$M_INHERLOG function modifier inhibits the logging of any error that occurs during the I/O operation.

9.3.2 Logical I/O Operations

In logical I/O operations, data is read from and written to logically addressable units of the device. Logical operations can be performed on both block-addressable and record-oriented devices. For block-addressable devices (such as disks), the addressable units are 512-byte blocks. They are numbered from 0 to $n-1$, where n is the number of blocks on the device. For record-oriented or non-block-structured devices (such as terminals), logical addressable units are not pertinent and are ignored. Logical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO).
- The issuing process has logical I/O privilege (LOG_IO).
- The volume is mounted foreign and the volume protection mask allows access to the device.

If none of these conditions is met, the logical I/O operation is rejected by the SYS$QIO system service, which returns a condition value of SS$_NOPRIV (no privilege). Figure 9–3 illustrates the logical I/O access checks in greater detail.

9.3.3 Virtual I/O Operations

You can perform virtual I/O operations on both record-oriented (non-file-structured) and block-addressable (file-structured) devices. For record-oriented devices (such as terminals), the virtual function is the same as a logical function; the virtual addressable units of the devices are ignored.

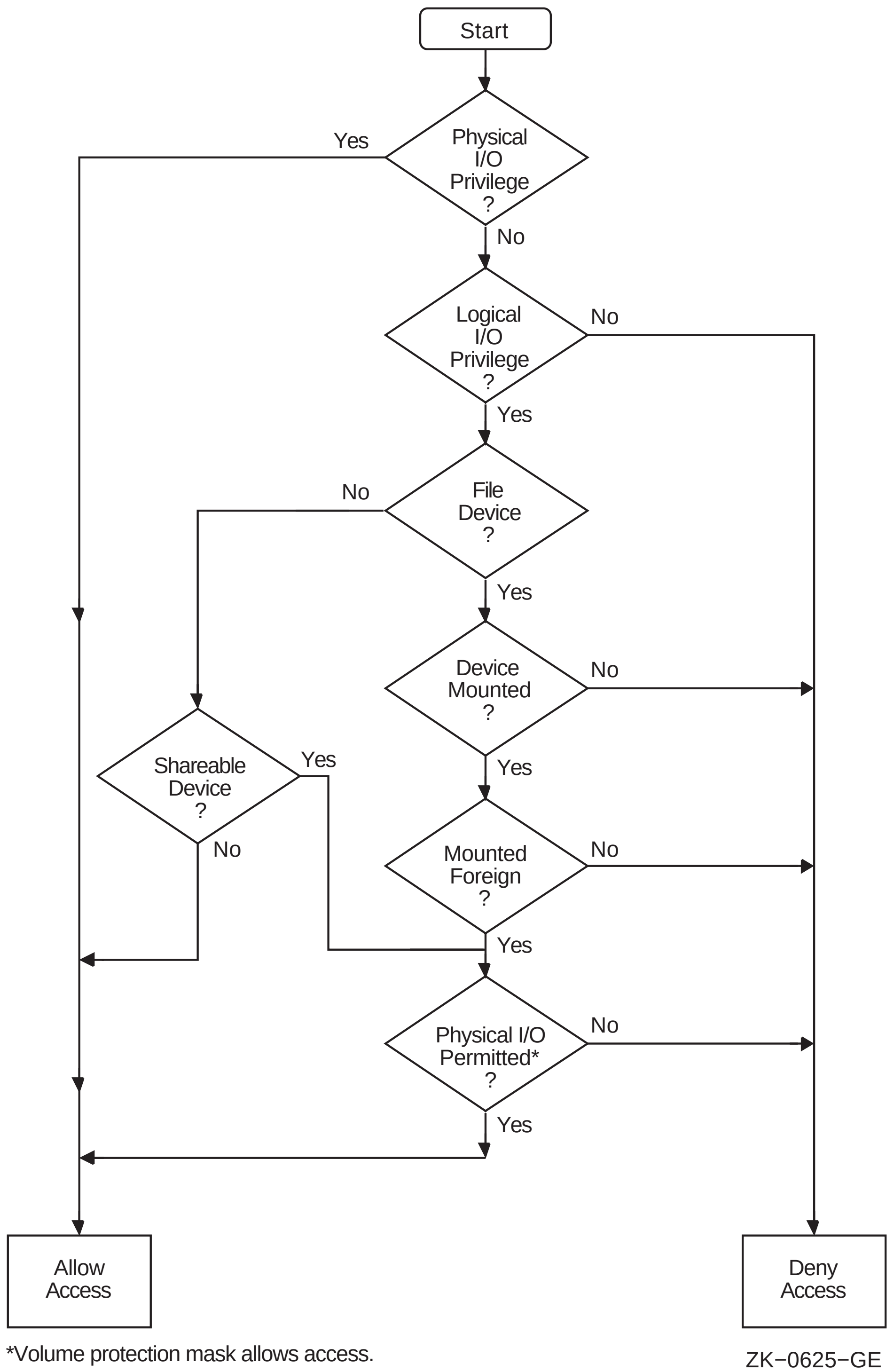
For block-addressable devices (such as disks), data is read from and written to open files. The addressable units in the file are 512-byte blocks. They are numbered starting at 1 and are relative to a file rather than to a device. Block-addressable devices must be mounted and structured and must contain a file that was previously accessed on the I/O channel.

Virtual I/O operations also require that the volume protection mask allow access to the device (a process having either physical or logical I/O privilege can override the volume protection mask). If these conditions are not met, the virtual I/O operation is rejected by the QIO system service, which returns one of the following condition values:

Condition Value	Meaning
SS$_NOPRIV	No privilege
SS$_DEVNOTMOUNT	Device not mounted
SS$_DEVFOREIGN	Volume mounted foreign

Figure 9–4 shows the relationship of physical, logical, and virtual I/O to the driver.

Figure 9–2 Physical I/O Access Checks



*Volume protection mask allows access.

Figure 9–3 Logical I/O Access Checks

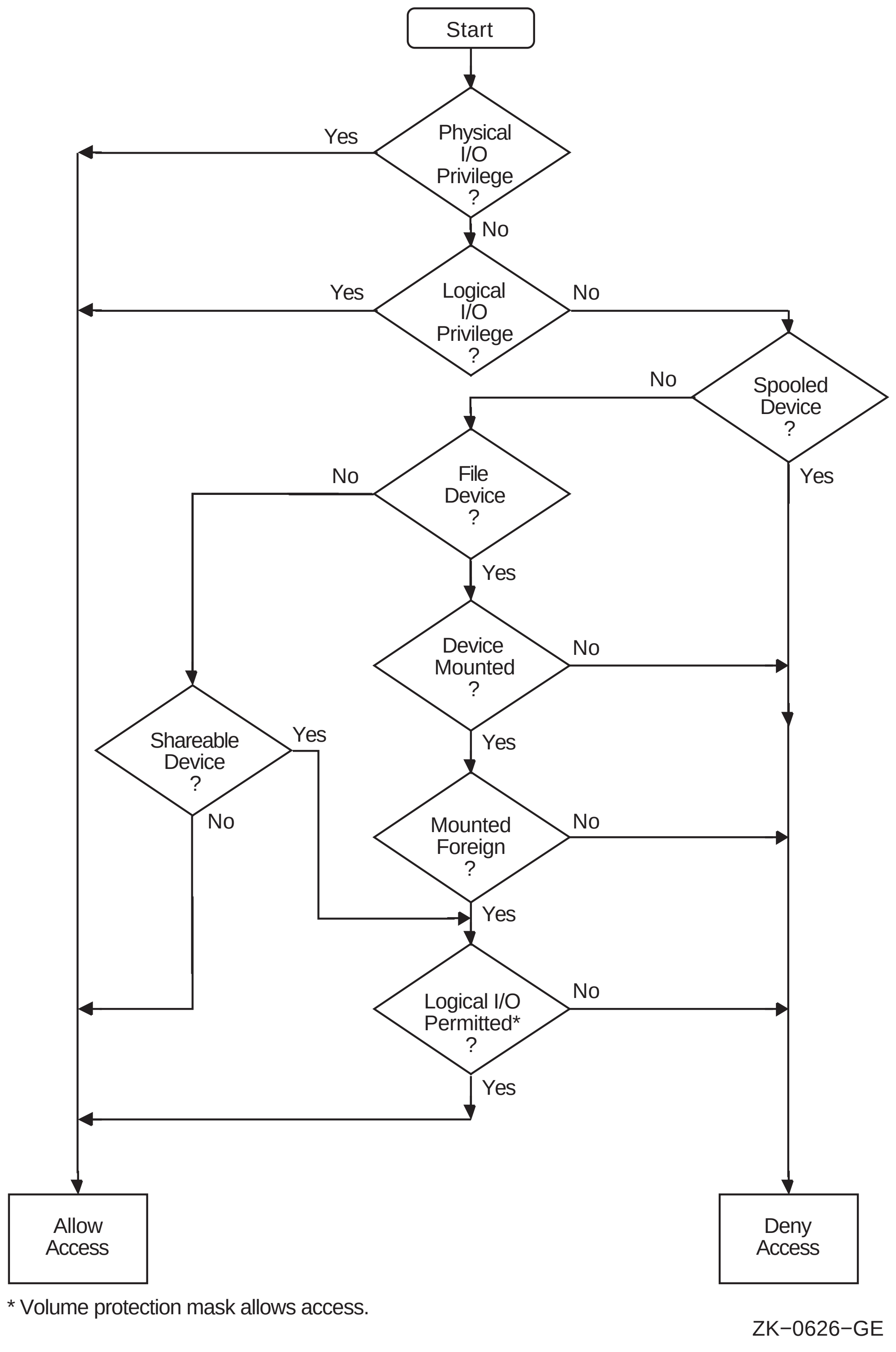
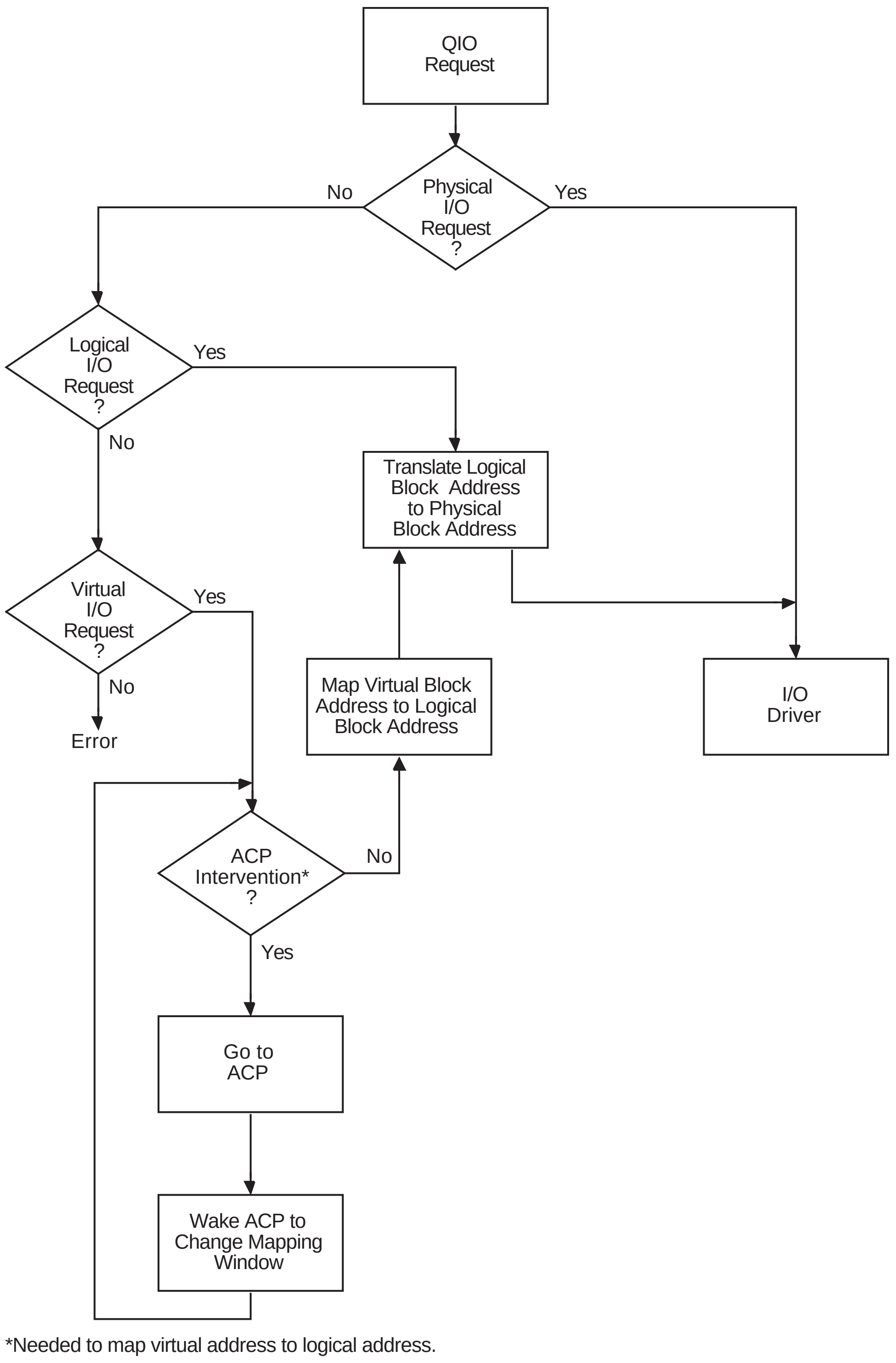


Figure 9–4 Physical, Logical, and Virtual I/O


QIO Request
Physical I/O Request ?
No
Yes
Logical I/O Request ?
Yes
No
Translate Logical Block Address to Physical Block Address
Virtual I/O Request ?
Yes
No
Error
Map Virtual Block Address to Logical Block Address
I/O Driver
ACP Intervention* ?
No
Yes
Go to ACP
Wake ACP to Change Mapping Window

*Needed to map virtual address to logical address.

ZK−0627−GE

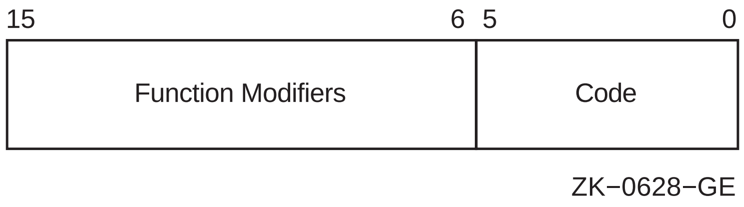
9.4 I/O Function Encoding

I/O functions fall into three groups that correspond to the three I/O device addressing modes (physical, logical, and virtual) described in Section 9.3. Depending on the device to which it is directed, an I/O function can be expressed in one, two, or all three modes.

I/O functions are described by 16-bit, symbolically expressed values that specify the particular I/O operation to be performed and any optional function modifiers. Figure 9–5 shows the format of the 16-bit function value.

Symbolic names for I/O function codes are defined by the $IODEF macro.

Figure 9–5 I/O Function Format



9.4.1 Function Codes

The low-order 6 bits of the function value are a code that specifies the particular operation to be performed. For example, the code for read logical block is expressed as IO$_READLBLK. Table 9–1 lists the symbolic values for read and write I/O functions in the three transfer modes.

Table 9–1 Read and Write I/O Functions

Physical I/O	Logical I/O	Virtual I/O
IO$_READPBLK	IO$_READLBLK	IO$_READVBLK
IO$_WRITEPBLK	IO$_WRITELBLK	IO$_WRITEVBLK

The set mode I/O function has a symbolic value of IO$_SETMODE.

Function codes are defined for all supported devices. Although some of the function codes (for example, IO$_READVBLK and IO$_WRITEVBLK) are used with several types of devices, most are device dependent; that is, they perform functions specific to particular types of devices. For example, IO$_CREATE is a device-dependent function code; it is used only with file-structured devices such as disks and magnetic tapes. The I/O user's reference documentation provides complete descriptions of the functions and function codes.

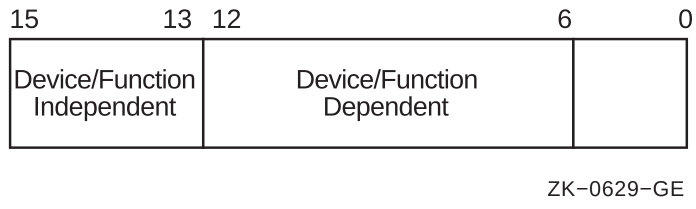
Note

You should determine the device class before performing any QIO function, because the requested function might be incompatible with some devices. For example, the SYS$INPUT device could be a terminal, a disk, or some other device. Unless this device is a terminal, an IO$_SETMODE request that enables a Ctrl/C AST is not performed.

9.4.2 Function Modifiers

The high-order 10 bits of the function value are function modifiers. These are individual bits that alter the basic operation to be performed. For example, you can specify the function modifier IO$M_NOECHO with the function IO$_READLBLK to a terminal. When used together, the two values are written in VAX MACRO as IO$_READLBLK!IO$M_NOECHO. This causes data typed at the terminal keyboard to be entered into the user buffer but not echoed to the terminal. Figure 9–6 shows the format of function modifiers.

Figure 9–6 Function Modifier Format



As shown in Figure 9–6, bits <15:13> are device- or function-independent bits, and bits <12:6> are device- or function-dependent bits. Device- or function-dependent bits have the same meaning, whenever possible, for different device classes. For example, the function modifier IO$M_ACCESS is used with both disk and magnetic tape devices to cause a file to be accessed during a create operation. Device- or function-dependent bits always have the same function within the same device class.

There are two device- or function-independent modifier bits: IO$M_INHRETRY and IO$M_DATACHECK (a third bit is reserved). IO$M_INHRETRY is used to inhibit all error recovery. If any error occurs and this modifier bit is specified, the operation is terminated immediately and a failure status is returned in the I/O status block (see Section 9.10). Use IO$M_DATACHECK to compare the data in memory with that on a disk or magnetic tape.

9.5 Assigning Channels

Before any input or output operation can be performed on a physical device, you must assign a channel to the device to provide a path between the process and the device. The Assign I/O Channel (SYS$ASSIGN) system service establishes this path.

When you write a call to the SYS$ASSIGN service, you must supply the name of the device, which can be a physical device name or a logical name, and the address of a word to receive the channel number. The service returns a channel number, and you use this channel number when you write an input or output request.

For example, the following lines assign an I/O channel to the device TTA2. The channel number is returned in the word at TTCHAN.

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

main() {
        unsigned int status;
        unsigned short ttchan;
        $DESCRIPTOR(ttname,"TTA2:");

/* Assign a channel to a device */
        status = SYS$ASSIGN( &ttname,   /* devnam - device name */
                             &ttchan,   /* chan - channel number */
                             0,         /* acmode - access mode */
                             0,         /* mbxnam - logical name for mailbox */
                             0 );       /* flags */
        if((status & 1) != 1)
                LIB$SIGNAL(status);

}
```

To assign a channel to the current default input or output device, use the logical name SYS$INPUT or SYS$OUTPUT.

For more details on how SYS$ASSIGN and other I/O services handle logical names, see Section 9.2.5.

9.6 Queuing I/O Requests

All input and output operations in the operating system are initiated with the Queue I/O Request (SYS$QIO) system service. The SYS$QIO system service permits direct interaction with the system's terminal driver. SYS$QIOs permit some operations that cannot be performed with language I/O statements and RTL routines; calls to SYS$QIO reduce overhead and permit asynchronous I/O operations. However, calls to SYS$QIO are device dependent. The SYS$QIO service queues the request and returns immediately to the caller. While the operating system processes the request, the program that issued the request can continue execution.

The format for SYS$QIO is as follows:

SYS$QIO([efn],chan,func[,iosb][,astadr][,astprm][,p1][,p2][,p3][,p4][,p5][,p6]

Required arguments to the SYS$QIO service include the channel number assigned to the device on which the I/O is to be performed, and a function code (expressed symbolically) that indicates the specific operation to be performed. Depending on the function code, one to six additional parameters may be required.

For example, the IO$_WRITEVBLK and IO$_READVBLK function codes are device-independent codes used to read and write single records or virtual blocks. These function codes are suitable for simple terminal I/O. They require parameters indicating the address of an input or output buffer and the buffer length. A call to SYS$QIO to write a line to a terminal may look like the following:

```
        unsigned int status, func=IO$_WRITEVBLK;
   .
   .
   .
        status = SYS$QIO(0,             /* efn - event flag */
                         ttchan,        /* chan - channel number */
                         func,          /* func - function modifier */
                         0,             /* iosb - I/O status block */
                         0,             /* astadr - AST routine */
                         0,             /* astprm - AST parameter */
                         buffadr,       /* p1 - output buffer */
                         buflen);       /* p2 - length of message */
```

Function codes are defined for all supported device types, and most of the codes are device dependent; that is, they perform functions specific to a particular device. The $IODEF macro defines symbolic names for these function codes. For information about how to obtain a listing of these symbolic names, see the *OpenVMS Programming Interfaces: Calling a System Routine*. For details about all function codes and an explanation of the parameters required by each, see the *OpenVMS I/O User's Reference Manual*.

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 by calling the SYS$ASSIGN system service, then use the assigned channel in the SYS$QIO or SYS$QIOW system service. To read from SYS$INPUT or write to SYS$OUTPUT, specify the appropriate logical name as the terminal name in the SYS$ASSIGN system service. In general, use SYS$QIO for asynchronous operations, and use SYS$QIOW for all other operations.

9.7 Synchronizing Service Completion

The SYS$QIO system service returns control to the calling program as soon as a request is queued; the status code returned in R0 indicates whether the request was queued successfully. To ensure proper synchronization of the queuing operation with respect to the program, the program must do the following:

- Test whether the operation was queued successfully.
- Test whether the operation itself completed successfully.

Optional arguments to the SYS$QIO service provide techniques for synchronizing I/O completion. There are three methods you can use to test for the completion of an I/O request:

- Specify the number of an event flag to be set when the operation completes.
- Specify the address of an AST routine to be executed when the operation completes.
- Specify the address of an I/O status block in which the system can place the return status when the operation completes.

  I/O status blocks are explained in Section 9.10.

The use of these three techniques is shown in the examples that follow. Example 9–1 shows specifying event flags.

Example 9–1 Event Flags

```
unsigned int status, efn=0, efn1=1, efn=2;
   .
   .
   .
status = SYS$QIO(efn1, . . .  );   /* Issue 1st I/O request */
if ((status & 1)) != 1)
LIB$SIGNAL( status );                  /* Queued successfully? */  1
   .
   .
   .
status = SYS$QIO(efn2, . . .  );   /* Issue second I/O request */  2
if ((status & 1)) != 1)                /* Queued successfully? */
        LIB$SIGNAL( status );
   .
   .
   .                                                              3
status = SYS$WFLAND( efn,              / *Wait until both are done */
                     &mask, . . .              4
                     .
                     .
                     .
```

1 When you specify an event flag number as an argument, SYS$QIO clears the event flag when it queues the I/O request. When the I/O completes, the flag is set.

2 In this example, the program issues two Queue I/O requests. A different event flag is specified for each request.

3 The Wait for Logical AND of Event Flags (SYS$WFLAND) system service places the process in a wait state until both I/O operations are complete. The **efn** argument indicates that the event flags are both in cluster 0; the **mask** argument indicates the flags for which the process is to wait.

4 Note that the SYS$WFLAND system service (and the other wait system services) wait for the event flag to be set; they do not wait for the I/O operation to complete. If some other event were to set the required event flags, the wait for event flag would complete too soon. You must coordinate the use of event flags carefully. (See Section 9.8 for a discussion of the recommended method for testing I/O completion.)

Example 9–2 shows specifying an AST routine.

Example 9–2 AST Routine

```
        unsigned int status, astprm=1;
   .
   .
   .
        status = SYS$QIO( . . .  &ttast,   /* I/O request with AST */ 1
                        astprm . . .  );
        if ((status & 1) != 1)                /* Queued successfully? */
                LIB$SIGNAL( status );
   .
   .
   .
}
void ttast ( int astprm ) {                   /* AST service routine */ 2
/* Handle I/O completion */
   .
   .
   .
        return;
}                              /* End of AST routine */
```

1 When you specify the **astadr** argument to the SYS$QIO system service, the system interrupts the process when the I/O completes and passes control to the specified AST service routine.

The SYS$QIO system service call specifies the address of the AST routine, TTAST, and a parameter to pass as an argument to the AST service routine. When $QIO returns control, the process continues execution.

2 When the I/O completes, the AST routine TTAST is called, and it responds to the I/O completion. By examining the AST parameter, TTAST can determine the origin of the I/O request.

When this routine is finished executing, control returns to the process at the point at which it was interrupted. If you specify the **astadr** argument in your call to SYS$QIO, you should also specify the **iosb** argument so that the AST routine can evaluate whether the I/O completed successfully.

Example 9–3 shows specifying an I/O status block.

Example 9–3 I/O Status Block

```
#include <stdio.h>
#include <ssdef.h>

   .
   .
   .
/* I/O  status block */
        struct {
                 unsigned short iostat, iolen;
                 unsigned int dev_info;
}ttiosb;                                                           1

        unsigned int status;
   .
   .
   .
        status = SYS$QIO(, . . .  , &ttiosb,  . . . );   2
        if ((status & 1) != 1)          /* Queued successfully? */
                LIB$SIGNAL( status );
   .
   .
   .
        while(ttiosb.iostat == 0) {
        /* Loop until done */                            3

        }
        if(ttiosb.iostat != SS$_NORMAL) {
        /* Perform error handling */
   .
   .
   .
        }
```

1 An I/O status block is a quadword structure that the system uses to post the status of an I/O operation. You must define the quadword area in your program. TTIOSB defines the I/O status block for this I/O operation. The **iosb** argument in the SYS$QIO system service refers to this quadword.

2 Instead of polling the low-order word of the I/O status block for the completion status, the program uses the preferred method of using an event flag and calling SYS$SYNCH to determine I/O completion.

3 The process polls the I/O status block. If the low-order word still contains zero, the I/O operation has not yet completed. In this example, the program loops until the request is complete.

9.8 Recommended Method for Testing Asynchronous Completion

Digital recommends that you use the Synchronize (SYS$SYNCH) system service to wait for completion of an asynchronous event. The SYS$SYNCH service correctly waits for the actual completion of an asynchronous event, even if some other event sets the event flag.

To use the SYS$SYNCH service to wait for the completion of an asynchronous event, you must specify both an event flag number and the address of an I/O status block (IOSB) in your call to the asynchronous system service. The asynchronous service queues the request and returns control to your program. When the asynchronous service completes, it sets the event flag and places the final status of the request in the IOSB.

In your call to SYS$SYNCH, you must specify the same **efn** and I/O status block that you specified in your call to the asynchronous service. The SYS$SYNCH service waits for the event flag to be set by means of the SYS$WAITFR system service. When the specified event flag is set, SYS$SYNCH checks the specified I/O status block. If the I/O status block is nonzero, the system service has completed and SYS$SYNCH returns control to your program. If the I/O status block is zero, SYS$SYNCH clears the event flag by means of the SYS$CLREF service and calls the $WAITFR service to wait for the event flag to be set.

The SYS$SYNCH service sets the event flag before returning control to your program. This ensures that the call to SYS$SYNCH does not interfere with testing for completion of another asynchronous event that completes at approximately the same time and uses the same event flag to signal completion.

The following call to the Queue I/O Request (SYS$QIO) system service demonstrates how the SYS$SYNCH service is used:

```
   .
   .
   .
        unsigned int status, event_flag = 1;
        struct {
                        short int iostat, iolen;
                        unsigned int dev_info;
}ttiosb;
   .
   .
   .
/* Request I/O */
        status = SYS$QIO (event_flag,  . . .  , &ttiosb  . . .  );
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
/* Wait until I/O completes */
        status = SYS$SYNCH (event_flag, &ttiosb );
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
```

Note

The SYS$QIOW service provides a combination of SYS$QIO and SYS$SYNCH.

9.9 Synchronous and Asynchronous Forms of Input/Output Services

You can execute some input/output services either synchronously or asynchronously. A “W” at the end of a system service name indicates the synchronous version of the system service.

The synchronous version of a system service combines the functions of the asynchronous version of the service and the Synchronize (SYS$SYNCH) system service. The synchronous version acts exactly as if you had used the asynchronous version of the system service followed immediately by a call to SYS$SYNCH; it queues the I/O request, and then places the program in a wait state until the I/O request completes. The synchronous version takes the same arguments as the asynchronous version.

Table 9–2 lists the asynchronous and synchronous names of input/output services that have synchronous versions.

Table 9–2 Asynchronous Input/Output Services and Their Synchronous Versions

Asynchronous Name	Synchronous Name	Description
$BRKTHRU	$BRKTHRUW	Breakthrough
$GETDVI	$GETDVIW	Get Device/Volume Information
$GETJPI	$GETJPIW	Get Job/Process Information
$GETLKI	$GETLKIW	Get Lock Information
$GETQUI	$GETQUIW	Get Queue Information
$GETSYI	$GETSYIW	Get Systemwide Information
$QIO	$QIOW	Queue I/O Request
$SNDJBC	$SNDJBCW	Send to Job Controller
$UPDSEC	$UPDSECW	Update Section File on Disk

9.9.1 Reading Operations with SYS$QIOW

The SYS$QIO and SYS$QIOW system services move one record of data from a terminal to a variable. For synchronous I/O, use SYS$QIOW. Complete information about the SYS$QIO and SYS$QIOW system services is presented in the *OpenVMS System Services Reference Manual*.

Note

Do not use the SYS$QIO and SYS$QIOW system services for input from a file or nonterminal device.

The SYS$QIO and SYS$QIOW system services place the data read in the variable specified in the **1** argument. The second word of the status block contains the offset from the beginning of the buffer to the terminator—hence, it equals the size of the data read. Always reference the data as a substring, using the offset to the terminator as the position of the last character (that is, the size of the substring). If you reference the entire buffer, your data will include the terminator for the operation (for example, the CR character) and any excess characters from a previous operation using the buffer. (The only exception to the substring guideline is if you deliberately overflow the buffer to terminate the I/O operation.)

Example 9–4 shows use of the SYS$QIOW system service and reads a line of data from the terminal and waits for the I/O to complete.

Example 9–4 Reading Data from the Terminal Synchronously

```
   .
   .
   .
INTEGER STATUS
! QIOW structures
INTEGER*2 INPUT_CHAN            ! I/O channel
INTEGER CODE,                   ! Type of I/O operation
2       INPUT_BUFF_SIZE,        ! Size of input buffer
2       PROMPT_SIZE,            ! Size of prompt
2       INPUT_SIZE              ! Size of input line as read
PARAMETER (PROMPT_SIZE = 13,
2          INPUT_BUFF_SIZE = 132)
CHARACTER*132 INPUT
CHARACTER*(*) PROMPT
PARAMETER (PROMPT = 'Input value: ')
! Define symbols used in I/O operations
INCLUDE '($IODEF)'
! Status block for QIOW
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,             ! Return status
2           TERM_OFFSET,        ! Location of line terminator
2           TERMINATOR,         ! Value of terminator
2           TERM_SIZE           ! Size of terminator
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Subprograms
INTEGER*4 SYS$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))
```

(continued on next page)

Example 9–4 (Cont.) Reading Data from the Terminal Synchronously

```
! 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
   .
   .
   .
```

9.9.2 Reading Operations with SYS$QIO

To perform an asynchronous read operation, use the SYS$QIO system service and specify an event flag (the first argument, which must be passed by value). Your program continues while the I/O is taking place. When you need the input from the I/O operation, invoke the SYS$SYNCH system service to wait for the event flag and status block specified in the SYS$QIO system service. If the I/O is not complete, your program pauses until it is. In this manner, you can overlap processing within your program. Naturally, you must take care not to assume data has been returned by the I/O operation before you call SYS$SYNCH and it returns successfully. Example 9–5 demonstrates an asynchronous read operation.

Example 9–5 Reading Data from the Terminal Asynchronously

```
   .
   .
   .
INTEGER STATUS
! QIO structures
INTEGER*2 INPUT_CHAN     ! I/O channel
INTEGER CODE,            ! Type of I/O operation
2       INPUT_BUFF_SIZE, ! Size of input buffer
2       PROMPT_SIZE,     ! Size of prompt
2       INPUT_SIZE       ! Size of input line as read
PARAMETER (INPUT_BUFF_SIZE = 132,
2          PROMPT = 13)
CHARACTER*132 INPUT
CHARACTER*(*) PROMPT
PARAMETER (PROMPT = 'Input value: ')
INCLUDE '($IODEF)'        ! Symbols used in I/O operations
! Status block for QIO
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,       ! Return status
2           TERM_OFFSET,  ! Location of line terminator
2           TERMINATOR,   ! Value of terminator
2           TERM_SIZE     ! Size of terminator
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
```

(continued on next page)

Example 9–5 (Cont.) Reading Data from the Terminal Asynchronously

```
! 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 check this status only after the I/O operation is complete (that is, after the call to SYS$SYNCH).

9.9.3 Write Operations with SYS$QIOW

The SYS$QIO and SYS$QIOW system services move one record of data from a character value to the terminal. Do not use these system services, as described here, for output to a file or nonterminal device.

For synchronous I/O, use SYS$QIOW and omit the first argument (the event flag number). For complete information about SYS$QIO and SYS$QIOW, refer to the *OpenVMS System Services Reference Manual*.

Example 9–6 writes a line of character data to the terminal.

Example 9–6 Writing Character Data to a Terminal

```
INTEGER STATUS,
2       ANSWER_SIZE
CHARACTER*31 ANSWER
INTEGER*2 OUT_CHAN
! Status block for QIO
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT,
2           BYTE_COUNT,
2           LINES_OUTPUT
  BYTE      COLUMN,
2           LINE
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Routines
INTEGER SYS$ASSIGN,
2       SYS$QIOW
! 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
```

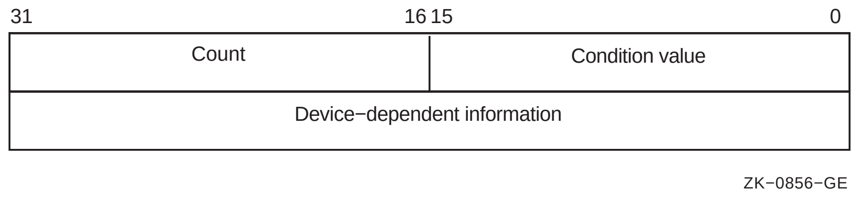
9.10 I/O Completion Status

When an I/O operation completes, the system posts the completion status in the I/O status block, if one is specified. The completion status indicates whether the operation completed successfully, the number of bytes that were transferred, and additional device-dependent return information.

Figure 9–7 illustrates the format for the SYS$QIO system service of the information written in the IOSB.

The first word contains a system status code indicating the success or failure of the operation. The status codes used are the same as for all returns from system services; for example, SS$_NORMAL indicates successful completion.

Figure 9–7 I/O Status Block



The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. For information about specific devices, see the *OpenVMS I/O User's Reference Manual*.

The second longword contains device-dependent return information.

System services other than SYS$QIO use the quadword I/O status block, but the format is different. See the description of each system service in the *OpenVMS System Services Reference Manual* for the format of the information written in the IOSB for that service.

To ensure successful I/O completion and the integrity of data transfers, you should check the IOSB following I/O requests, particularly for device-dependent I/O functions. For complete details about how to use the I/O status block, see the *OpenVMS I/O User's Reference Manual*.

9.11 Deassigning I/O Channels

When a process no longer needs access to an I/O device, it should release the channel assigned to the device by calling the Deassign I/O Channel (SYS$DASSGN) system service:

```
$DASSGN_S CHAN=TTCHAN
```

This service call releases the terminal channel assignment acquired in the SYS$ASSIGN example shown in Section 9.5. The system automatically deassigns channels for a process when the image that assigned the channel exits.

9.12 Using Complete Terminal I/O

The following example shows a complete sequence of input and output operations using the $QIOW macro to read and write lines to the current default SYS$INPUT device. Because the input/output of this program must be to the current terminal, it functions correctly only if you execute it interactively.

```
#include <stdio.h>
#include <ssdef.h>
#include <descrip.h>
#include <string.h>
#include <iodef.h>

/* I/O status block */
struct {                                                           1
        unsigned short iostat, ttiolen;
        unsigned int dev_info;
}ttiosb;
```

```
main() {
        unsigned int status ,outlen, inlen;
        unsigned short ttchan;
        char buffer[80];                                          2
        $DESCRIPTOR(ttname,"SYS$INPUT");                          3

/* Assign a channel */
        status = SYS$ASSIGN(&ttname,    /* devnam - device number */ 4
                        &ttchan,        /* chan - channel number */
                        0, 0, 0);
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

/* Request I/O */
        inlen = strlen(buffer);
        status = SYS$QIOW(0,                    /* efn - event flag */
                        ttchan,                 /* chan - channel number */
                        IO$_READVBLK,           /* func - function modifier */
                        &ttiosb,                /* iosb - I/O status block */
                        0,                      /* astadr - AST routine */
                        0,                      /* astprm - AST parameter */
                        buffer,                 /* p1 - buffer */
                        inlen,                  /* p2 - length of buffer */
                        0, 0, 0, 0);    5
        if ((status & 1) != 1)          6
                LIB$SIGNAL( status );

/* Get length from IOSB */
        outlen = ttiosb.ttiolen;        7

status = SYS$QIOW(0, ttchan, IO$_WRITEVBLK, &ttiosb, 0, 0, buffer, outlen,
                0, 0, 0, 0);
        if ((status & 1) != 1)
                LIB$SIGNAL( status );   8

/* Deassign the channel */
        status = SYS$DASSGN( ttchan ); /* chan - channel */  9
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

}
```

1 The IOSB for the I/O operations is structured so that the program can easily check for the completion status (in the first word) and the length of the input string returned (in the second word).

2 The string will be read into the buffer BUFFER; the longword OUTLEN will contain the length of the string for the output operation.

3 The TTNAME label is a character string descriptor for the logical device SYS$INPUT, and TTCHAN is a word to receive the channel number assigned to it.

4 The $ASSIGN service assigns a channel and writes the channel number at TTCHAN.

5 If the $ASSIGN service completes successfully, the $QIOW macro reads a line from the terminal, and requests that the completion status be posted in the I/O status block defined at TTIOSB.

6 The process waits until the I/O is complete, then checks the first word in the I/O status block for a successful return. If unsuccessful, the program takes an error path.

7 The length of the string read is moved into the longword at OUTLEN, because the $QIOW macro requires a longword argument. However, the length field of the I/O status block is only 1 word long. The $QIOW macro writes the line just read to the terminal.

8 The program performs error checks. First, it ensures that the $OUTPUT macro successfully queued the I/O request; then, when the request is completed, it ensures that the I/O was successful.

9 When all I/O operations on the channel are finished, the channel is deassigned.

9.13 Canceling I/O Requests

If a process must cancel I/O requests that have been queued but not yet completed, it can issue the Cancel I/O On Channel (SYS$CANCEL) system service. All pending I/O requests issued by the process on that channel are canceled; you cannot specify a particular I/O request.

The SYS$CANCEL system service performs an asynchronous cancel operation. This means that the application *must* wait for each I/O operation issued to the driver to complete before checking the status for that operation.

For example, you can call the SYS$CANCEL system service as follows:

```
        unsigned int status, efn1=3, efn2=4;
   .
   .
   .
        status = SYS$QIO(efn1, ttchan, &iosb1,  . . .  );
        status = SYS$QIO(efn2, ttchan, &iosb2,  . . .  );
   .
   .
   .
        status = SYS$CANCEL(ttchan);
        status = SYS$SYNCH(efn1, &iosb1);
        status = SYS$SYNCH(efn2, &iosb2);
```

In this example, the SYS$CANCEL system service initiates the cancellation of all pending I/O requests to the channel whose number is located at TTCHAN.

The SYS$CANCEL system service returns after initiating the cancellation of the I/O requests. If the call to SYS$QIO specified an event flag, AST service routine, or I/O status block, the system sets the flag, delivers the AST, or posts the I/O status block as appropriate when the cancellation is completed.

9.14 Logical Names and Physical Device Names

When you specify a device name as input to an I/O system service, it can be a physical device name or a logical name. If the device name contains a colon (:), the colon and the characters after it are ignored. When an underscore character (_) precedes a device name string, it indicates that the string is a physical device name string, for example, _TTB3:.

Any string that does not begin with an underscore is considered a logical name, even though it may be a physical device name. Table 9–3 lists system services that translate a logical name iteratively until a physical device name is returned, or until the system default number of translations have been performed.

Table 9–3 System Services for Translating Logical Names

System Service	Definition
SYS$ALLOC	Allocate Device
SYS$ASSIGN	Assign I/O Channel
SYS$BRDCST	Broadcast
SYS$DALLOC	Deallocate Device
SYS$DISMOU	Dismount Volume
SYS$GETDEV	Get I/O Device Information
SYS$GETDVI	Get Device/Volume Information
SYS$MOUNT	Mount Volume

In each translation, the logical name tables defined by the logical name LNM$FILE_DEV are searched in order. These tables, listed in search order, are normally LNM$PROCESS, LNM$JOB, LNM$GROUP, and LNM$SYSTEM. If a physical device name is located, the I/O request is performed for that device.

If the services do not locate an entry for the logical name, the I/O service treats the name specified as a physical device name. When you specify the name of an actual physical device in a call to one of these services, include the underscore character to bypass the logical name translation.

When the SYS$ALLOC system service returns the device name of the physical device that has been allocated, the device name string returned is prefixed with an underscore character. When this name is used for the subsequent SYS$ASSIGN system service, the SYS$ASSIGN service does not attempt to translate the device name.

If you use logical names in I/O service calls, you must be sure to establish a valid device name equivalence before program execution. You can do this by issuing a DEFINE command from the command stream, or by having the program establish the equivalence name before the I/O service call with the Create Logical Name (SYS$CRELNM) system service.

For details about how to create and use logical names, see Chapter 10.

9.15 Device Name Defaults

If, after logical name translation, a device name string in an I/O system service call does not fully specify the device name (that is, device, controller, and unit), the service either provides default values for nonspecified fields, or provides values based on device availability.

The following rules apply:

- The SYS$ASSIGN and SYS$DALLOC system services apply default values, as shown in Table 9–4.
- The SYS$ALLOC system service treats the device name as a generic device name and attempts to find a device that satisfies the components of the device name specified, as shown in Table 9–4.

Table 9–4 Default Device Names for I/O Services

Device	Device Name[1]	Generic Device
dd:	*dd*A0: (unit 0 on controller A)	*ddxy*: (any available device of the specified type)
ddc:	*ddc*0: (unit 0 on controller specified)	*ddcy*: (any available unit on the specified controller)
ddu:	*dd*A*u*: (unit specified on controller A)	*ddxu*: (device of specified type and unit on any available controller)
ddcu:	*ddcu*: (unit and controller specified)	*ddcu*: (unit and controller specified)

[1]See the *OpenVMS User's Manual* for a summary of the device names.

Key

dd—Specified device type (capital letters indicate a specific controller; numbers indicate a specific unit)
c—Specified controller
x—Any controller
u—Specified unit number
y—Any unit number

9.16 Obtaining Information About Physical Devices

The Get Device/Volume Information (SYS$GETDVI) system service returns information about devices. The information returned is specified by an item list created before the call to SYS$GETDVI.

When you call the SYS$GETDVI system service, you must provide the address of an item list that specifies the information to be returned. The format of the item list is described in the description of SYS$GETDVI in the *OpenVMS System Services Reference Manual*. The *OpenVMS I/O User's Reference Manual* contains details on the device-specific information these services return.

In cases where a generic (that is, nonspecific) device name is used in an I/O service, a program may need to find out what device has been used. To do this, the program should provide SYS$GETDVI with the number of the channel to the device and request the name of the device with the DVI$_DEVNAM item identifier.

The operating system also supports a device called the null device for program development. The mnemonic for the null device is NL. Its characteristics are as follows:

- A read from NL returns an end-of-file error (SS$_ENDOFFILE).
- A write to NL immediately returns a success message (SS$_NORMAL).

The null device functions as a virtual device to which you can direct output but from which the data does not return.

9.16.1 Checking the Terminal Device

You are restricted to a terminal device if you use any of the special functions described in this section. If the user of your program redirects SYS$INPUT or SYS$OUTPUT to a file or nonterminal device, an error occurs. You can use the SYS$GETDVIW system service to make sure the logical name is associated with a terminal, as shown in Example 9–7. SYS$GETDVIW returns a status of SS$_IVDEVNAM if the logical name is defined as a file or otherwise does not equate to a device name. The type of device is the response associated with the DVI$_DEVCLASS request code and should be DC$_TERM for a terminal.

Example 9–7 Using SYS$GETDVIW to Verify the Device Name

```
RECORD /ITMLST/ DVI_LIST
LOGICAL*4 STATUS
! GETDVI buffers
INTEGER CLASS,             ! Response buffer
2       CLASS_LEN          ! Response length
! GETDVI symbols
INCLUDE '($DCDEF)'
INCLUDE '($SSDEF)'
INCLUDE '($DVIDEF)'
! Define subprograms
INTEGER SYS$GETDVIW
! 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
```

9.16.2 Terminal Characteristics

The *OpenVMS I/O User's Reference Manual* describes device-specific characteristics associated with terminals. To examine a characteristic, issue a call to SYS$QIO or SYS$QIOW system service with the IO$_SENSEMODE function and examine the appropriate bit in the structure returned to the **P1** argument. To change a characteristic:

1. Issue a call to SYS$QIO or SYS$QIOW system service with the IO$_SENSEMODE function.
2. Set or clear the appropriate bit in the structure returned to the **P1** argument.
3. Issue a call to SYS$QIO or SYS$QIOW system service with the IO$_SETMODE function passing, as the **P1** argument, to modify the structure you obtained from the sense mode operation.

Example 9–8 turns off the HOSTSYNC terminal characteristic. To check whether NOHOSTSYNC has been set, enter the SHOW TERMINAL command.

Example 9–8 Disabling the HOSTSYNC Terminal Characteristic

```
   .
   .
   .
INTEGER*4 STATUS
! I/O channel
INTEGER*2 INPUT_CHAN
! I/O status block
STRUCTURE /IOSTAT_BLOCK/
  INTEGER*2 IOSTAT
  BYTE      TRANSMIT,
2           RECEIVE,
2           CRFILL,
2           LFFILL,
2           PARITY,
2           ZERO
END STRUCTURE
RECORD /IOSTAT_BLOCK/ IOSB
! Characteristics buffer
! Note: basic characteristics are first three
!       bytes of second longword -- length is
!       last byte
STRUCTURE /CHARACTERISTICS/
  BYTE      CLASS,
2           TYPE
  INTEGER*2 WIDTH
  UNION
   MAP
    INTEGER*4 BASIC
   END MAP
   MAP
    BYTE LENGTH(4)
   END MAP
  END UNION
  INTEGER*4 EXTENDED
END STRUCTURE
RECORD /CHARACTERISTICS/ CHARBUF
! Define symbols used for I/O and terminal operations
INCLUDE '($IODEF)'
INCLUDE '($TTDEF)'
! Subroutines
INTEGER*4 SYS$ASSIGN,
2         SYS$QIOW
! 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)
```

(continued on next page)

Example 9–8 (Cont.) Disabling the HOSTSYNC Terminal Characteristic

```
! 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. (Resetting is not necessary if you just use modifiers on each read operation.) To ensure that the restoration is performed if the program aborts (for example, if the user presses Ctrl/Y), you should restore the user's environment in an exit handler. See Chapter 13 for a description of exit handlers.

9.16.3 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 **4** argument in the read QIO operation. You can examine the terminator that ended the read operation by examining the input buffer starting at the terminator offset (second word of the I/O status block). The length, in bytes, of the terminator is specified by the high-order word of the I/O status block. The third word of the I/O status block contains the value of the first character of the terminator.

Examining the terminator enables you to read escape sequences from the terminal, provided that you modify the QIO read operation with the IO$M_ ESCAPE modifier (or the ESCAPE terminal characteristic is set). The first character of the terminator will be the ESC character (an ASCII value of 27). The remaining characters will contain the value of the escape sequence.

9.16.4 File Terminators

You must examine the terminator to detect end-of-file (Ctrl/Z) on the terminal. No error condition is generated at the QIO level. If the user presses Ctrl/Z, the terminator will be the SUB character (an ASCII value of 26).

9.17 Device Allocation

Many I/O devices are shareable; that is, more than one process at a time can access the device. By calling the Assign I/O Channel (SYS$ASSIGN) system service, a process is given a channel to the device for I/O operations.

In some cases, a process may need exclusive use of a device so that data is not affected by other processes. To reserve a device for exclusive use, you must allocate it.

Device allocation is normally accomplished with the DCL command ALLOCATE. A process can also allocate a device by calling the Allocate Device (SYS$ALLOC) system service. When a device has been allocated by a process, only the process that allocated the device and any subprocesses it creates can assign channels to the device.

When you call the SYS$ALLOC system service, you must provide a device name. The device name specified can be any of the following:

- A physical device name; for example, the tape drive MTB3:
- A logical name; for example, TAPE
- A generic device name; for example, MT:

If you specify a physical device name, SYS$ALLOC attempts to allocate the specified device.

If you specify a logical name, SYS$ALLOC translates the logical name and attempts to allocate the physical device name equated to the logical name.

If you specify a generic device name (that is, if you specify a device type but do not specify a controller or unit number, or both), SYS$ALLOC attempts to allocate any device available of the specified type. For more information about the allocation of devices by generic names, see Section 9.15.

When you specify generic device names, you must provide fields for the SYS$ALLOC system service to return the name and the length of the physical device that is actually allocated so that you can provide this name as input to the SYS$ASSIGN system service.

The following example illustrates the allocation of a tape device specified by the logical name TAPE:

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

main() {
        unsigned int status;
        char devstr[64];
        unsigned short phylen, tapechan;

        $DESCRIPTOR(logdev,"TAPE");     /* Descriptor for logical name */
        $DESCRIPTOR(devdesc,devstr);    /* Descriptor for physical name */

/* Allocate a device */
        status = SYS$ALLOC( &logdev,    /* devnam - device name */      1
                            &phylen,    /* phylen - length device name string */
                            &devdesc,   /* phybuf - buffer for devnam string */
                            0, 0);
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

/* Assign a channel to the device */
        status = SYS$ASSIGN( &devdesc,          /* devnam - device name */  2
                             &tapechan,         /* chan - channel number */
                             0, 0, 0);
        if ((status & 1) != 1)
                LIB$SIGNAL( status );

/* Deassign the channel */
        status = SYS$DASSGN( tapechan );        /* chan - channel number */3
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
/* Deallocate the device */
        status = SYS$DALLOC( &devdesc,          /* devnam - device name */
                             0 );               /* acmode - access mode */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
}
```

1 The SYS$ALLOC system service call requests allocation of a device corresponding to the logical name TAPE, defined by the character string descriptor LOGDEV. The argument DEVDESC refers to the buffer provided to receive the physical device name of the device that is allocated and the length of the name string. The SYS$ALLOC service translates the logical name TAPE and returns the equivalence name string of the device actually allocated into the buffer at DEVDESC. It writes the length of the string in the first word of DEVDESC.

2 The SYS$ASSIGN command uses the character string returned by the SYS$ALLOC system service as the input device name argument, and requests that the channel number be written into TAPECHAN.

3 When I/O operations are completed, the SYS$DASSGN system service deassigns the channel, and the SYS$DALLOC system service deallocates the device. The channel must be deassigned before the device can be deallocated.

9.17.1 Implicit Allocation

Devices that cannot be shared by more than one process (for example, terminals and line printers) do not have to be explicitly allocated. Because they are nonshareable, they are implicitly allocated by the SYS$ASSIGN system service when SYS$ASSIGN is called to assign a channel to the device.

9.17.2 Deallocation

When the program has finished using an allocated device, it should release the device with the Deallocate Device (SYS$DALLOC) system service to make it available for other processes.

At image exit, the system automatically deallocates devices allocated by the image.

9.18 Mounting, Dismounting, and Initializing Volumes

This section introduces you to using system services to mount, dismount, and initialize disk and tape volumes.

9.18.1 Mounting a Volume

Mounting a volume establishes a link between a volume, a device, and a process. A volume, or volume set, must be mounted before I/O operations can be performed on the volume. You interactively mount or dismount a volume from the DCL command stream with the MOUNT or DISMOUNT command. A process can also mount or dismount a volume or volume set programmatically using the Mount Volume (SYS$MOUNT) or the Dismount Volume (SYS$DISMOU) system service, respectively.

Mounting a volume involves two operations:

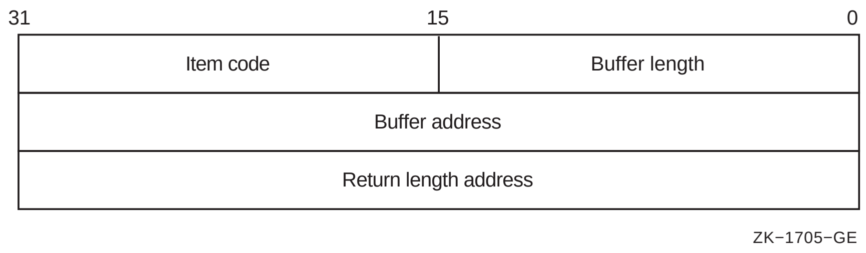
1. Place the volume on the device and start the device (by pressing the START or LOAD button).
2. Mount the volume with the $MOUNT system service.

9.18.1.1 Calling the SYS$MOUNT System Service

The Mount Volume (SYS$MOUNT) system service allows a process to mount a single volume or a volume set. When you call the SYS$MOUNT system service, you must specify a device name.

The SYS$MOUNT system service has a single argument, which is the address of a list of item descriptors. The list is terminated by a longword of binary zeros. Figure 9–8 shows the format of an item descriptor.

Figure 9–8 SYS$MOUNT Item Descriptor



Most item descriptors do not have to be in any order. To mount volume sets, you must specify one item descriptor per device and one item descriptor per volume; you must specify the descriptors for the volumes in the same order as the descriptors for the devices on which the volumes are loaded.

For item descriptors other than device and volume names, if you specify the same item descriptor more than once, the last occurrence of the descriptor is used.

The following example illustrates a call to SYS$MOUNT. The call is equivalent to the DCL command that precedes the example.

```
$ MOUNT/SYSTEM/NOQUOTA  DRA4:,DRA5:  USER01,USER02  USERD$


#include <stdio.h>
#include <descrip.h>
#include <mntdef.h>
   .
   .
   .

struct {
        unsigned short buflen, item_code;
        void *bufaddr;
        int *retlenaddr;
}itm;

struct {
         struct itm itm[6];
         unsigned int terminator;
}itm_lst ;

main() {
   .
   .
   .
        unsigned int status, flags;
        $DESCRIPTOR(dev1,"DRA4:");
        $DESCRIPTOR(vol1,"USER01:");
        $DESCRIPTOR(dev2,"DRA5:");
        $DESCRIPTOR(vol2,"USER02:");
        $DESCRIPTOR(log,"USERD$:");

flags = MNT$M_SYSTEM | MNT$M_NODISKQ;

itm_lst.itm[0].buflen = 4;
itm_lst.itm[0].item_code = MNT$_FLAGS;
itm_lst.itm[0].bufaddr = flags;
itm_lst.itm[0].retlenaddr = 0;

itm_lst.itm[1].buflen = 5;
itm_lst.itm[1].item_code = MNT$_DEVNAM;
itm_lst.itm[1].bufaddr = dev1;
itm_lst.itm[1].retlenaddr = 0;

itm_lst.itm[2].buflen = 6;
itm_lst.itm[2].item_code = MNT$_VOLNAM;
itm_lst.itm[2].bufaddr = vol1;
itm_lst.itm[2].retlenaddr = 0;

itm_lst.itm[3].buflen = 5;
itm_lst.itm[3].item_code = MNT$_DEVNAM;
itm_lst.itm[3].bufaddr = dev2;
itm_lst.itm[3].retlenaddr = 0;

itm_lst.itm[4].buflen = 6;
itm_lst.itm[4].item_code = MNT$_VOLNAM;
itm_lst.itm[4].bufaddr = vol2;
itm_lst.itm[4].retlenaddr = 0;

itm_lst.itm[5].buflen = 6;
itm_lst.itm[5].item_code = MNT$_LOGNAM;
itm_lst.itm[5].bufaddr = log;
itm_lst.itm[5].retlenaddr = 0;

itm_lst.terminator = 0;
   .
   .
   .
        status = SYS$MOUNT ( &itm_lst );
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
}
```

9.18.1.2 Calling the SYS$DISMOU System Service

The SYS$DISMOU system service allows a process to dismount a volume or volume set. When you call SYS$DISMOU, you must specify a device name. If the volume mounted on the device is part of a fully mounted volume set, and you do not specify flags, the whole volume set is dismounted.

The following example illustrates a call to SYS$DISMOU. The call dismounts the volume set mounted in the previous example.

```
    $DESCRIPTOR(dev1_desc,"DRA4:");
   .
   .
   .
        status = SYS$DISMOU(&dev1_desc); /* devnam - device */
   .
   .
   .
```

9.18.2 Initializing Volumes

Initializing a volume writes a label on the volume, sets protection and ownership for the volume, formats the volume (depending on the device type), and overwrites data already on the volume.

You interactively initialize a volume from the DCL command stream using the INITIALIZE command. A process can programmatically initialize a volume using the Initialize Volume (SYS$INIT_VOL) system service.

9.18.2.1 Calling the Initialize Volume System Service

You must specify a device name and a new volume name when you call the SYS$INIT_VOL system service. You can also use the **itmlst** argument of $INIT_VOL to specify options for the initialization. For example, you can specify that data compaction should be performed by specifying the INIT$_COMPACTION item code. See the *OpenVMS System Services Reference Manual* for more information on initialization options.

Before initializing the volume with SYS$INIT_VOL, be sure you have placed the volume on the device and started the device (by pressing the START or LOAD button).

The default format for files on disk volumes is called Files–11 On-Disk Structure Level 2. Files–11 On-Disk Structure Level 1 format, available on VAX systems, is used by other Digital operating systems, including RSX–11M, RSX–11M–PLUS, RSX–11D, and IAS, but is not supported on Alpha systems. For more information, see the *OpenVMS System Manager's Manual*.

Here are two examples of calling SYS$INIT_VOL programmatically: one from a C program and one from a BASIC program.

The following example illustrates a call to SYS$INIT_VOL from DEC C:

```
#include <descrip.h>
#include <initdef.h>

struct item_descrip_3
{
    unsigned short buffer_size;
    unsigned short item_code;
    void *buffer_address;
    unsigned short *return_length;
};

main ()
{
    unsigned long
        density_code,
        status;
    $DESCRIPTOR(drive_dsc, "MUA0:");
    $DESCRIPTOR(label_dsc, "USER01");
    struct
    {
        struct item_descrip_3 density_item;
        long terminator;
    } init_itmlst;

    /*
    ** Initialize the input item list.
    */
    density_code = INIT$K_DENSITY_6250_BPI;
    init_itmlst.density_item.buffer_size = 4;
    init_itmlst.density_item.item_code = INIT$_DENSITY;
    init_itmlst.density_item.buffer_address = &density_code;

    init_itmlst.terminator = 0;

    /*
    ** Initialize the volume.
    */

    status = SYS$INIT_VOL (&drive_dsc, &label_dsc, &init_itmlst);

    /*
    ** Report an error if one occurred.
    */

    if ((status & 1) != 1)
        LIB$STOP (status);
}
```

The following example illustrates a call to SYS$INIT_VOL from VAX BASIC:

```
OPTION TYPE = EXPLICIT

%INCLUDE '$INITDEF' %FROM %LIBRARY

EXTERNAL LONG FUNCTION SYS$INIT_VOL

RECORD ITEM_DESC
        VARIANT
        CASE
            WORD BUFLEN
            WORD ITMCOD
            LONG BUFADR
            LONG LENADR
        CASE
            LONG TERMINATOR
        END VARIANT
END RECORD

DECLARE LONG RET_STATUS, &
    ITEM_DESC INIT_ITMLST(2)

! Initialize the input item list.

INIT_ITMLST(0)::ITMCOD = INIT$_READCHECK
INIT_ITMLST(1)::TERMINATOR = 0

! Initialize the volume.

RET_STATUS = SYS$INIT_VOL ("DJA21:" BY DESC, "USERVOLUME" BY DESC,
INIT_ITMLST() BY REF)
```

9.19 Formatting Output Strings

When you are preparing output strings for a program, you may need to insert variable information into a string prior to output, or you may need to convert a numeric value to an ASCII string. The Formatted ASCII Output (SYS$FAO) system service performs these functions.

Input to the SYS$FAO system service consists of the following:

- A control string that contains the fixed text portion of the output and formatting directives. The directives indicate the position within the string where substitutions are to be made, and describe the data type and length of the input values that are to be substituted or converted.

- An output buffer to contain the string after conversions and substitutions have been made.
- An optional argument indicating a word to receive the final length of the formatted output string.
- Parameters that provide arguments for the formatting directives.

The following example shows a call to the SYS$FAO system service to format an output string for a SYS$QIOW macro. Complete details on how to use SYS$FAO, with additional examples, are provided in the description of the SYS$FAO system service in the *OpenVMS System Services Reference Manual*.

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

main() {

        unsigned int status, faolen;
        char faobuf[80];
        $DESCRIPTOR(faostr,"FILE !AS DOES NOT EXIST");    1
        $DESCRIPTOR(outbuf, faobuf);                      2
        $DESCRIPTOR(filespec,"DISK$USER:MYFILE.DAT");     3

        status = SYS$FAO( &faostr, &outlen, &outbuf, &filespec );  4
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        status = SYS$QIOW(  . . .  faobuf, outlen,  . . .   ); 5
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
}
```

1 FAOSTR provides the FAO control string. !AS is an example of an FAO directive: it requires an input parameter that specifies the address of a character string descriptor. When SYS$FAO is called to format this control string, !AS will be substituted with the string whose descriptor address is specified.

2 FAODESC is a character string descriptor for the output buffer; SYS$FAO will write the string into the buffer, and will write the length of the final formatted string in the low-order word of FAOLEN. (A longword is reserved so that it can be used for an input argument to the SYS$QIOW macro.)

3 FILESPEC is a character string descriptor defining an input string for the FAO directive !AS.

4 The call to SYS$FAO specifies the control string, the output buffer and length fields, and the parameter P1, which is the address of the string descriptor for the string to be substituted.

5 When SYS$FAO completes successfully, SYS$QIOW writes the following output string:

```
FILE DISK$USER:MYFILE.DAT DOES NOT EXIST
```

9.20 Mailboxes

Mailboxes are virtual devices that can be used for communication among processes. You accomplish actual data transfer by using OpenVMS RMS or I/O services. When the Create Mailbox and Assign Channel (SYS$CREMBX) system service creates a mailbox, it also assigns a channel to it for use by the creating process. Other processes can then assign channels to the mailbox using either the SYS$CREMBX or SYS$ASSIGN system service.

The SYS$CREMBX system service creates the mailbox. The SYS$CREMBX system service identifies a mailbox by a user-specified logical name and assigns it an equivalence name. The equivalence name is a physical device name in the format MBA*n*, where *n* is a unit number. The equivalence name has the terminal attribute.

When another process assigns a channel to the mailbox with the SYS$CREMBX or SYS$ASSIGN system service, it can identify the mailbox by its logical name. The service automatically translates the logical name. The process can obtain the MBA*n* name by translating the logical name (with the SYS$TRNLNM system service), or it can call the Get Device/Volume Information (SYS$GETDVI) system service to obtain the unit number and the physical device name.

VAX

On VAX systems, channels assigned to mailboxes can be either bidirectional or unidirectional. Bidirectional channels (read/write) allow both SYS$QIO read and SYS$QIO write requests to be issued to the channel. Unidirectional channels (read-only or write-only) allow only a read request or a write request to the channel. The unidirectional channels and unidirectional $QIO function modifiers provide for greater synchronization between users of the mailbox.

The Create Mailbox and Assign Channel (SYS$CREMBX) and Assign I/O Channel (SYS$ASSIGN) system services use the **flags** argument to enable unidirectional channels. If the **flags** argument is not specified or is zero, then the channel assigned to the mailbox is bidirectional (read/write). For more information, see the discussion and programming examples in the mailbox driver chapter in the *OpenVMS I/O User's Reference Manual*. Chapter 2 of this manual also discusses the use of mailboxes. ♦

Mailboxes are either temporary or permanent. You need the user privileges TMPMBX and PRMMBX to create temporary and permanent mailboxes, respectively.

For a temporary mailbox, the SYS$CREMBX service enters the logical name and equivalence name in the logical name table LNM$TEMPORARY_MAILBOX. This logical name table name usually specifies the LNM$JOB logical name table name. The system deletes a temporary mailbox when no more channels are assigned to it.

For a permanent mailbox, the SYS$CREMBX service enters the logical name and equivalence name in the logical name table LNM$PERMANENT_MAILBOX. This logical name table name usually specifies the LNM$SYSTEM logical name table name. Permanent mailboxes continue to exist until they are specifically marked for deletion with the Delete Mailbox (SYS$DELMBX) system service.

The following example shows how processes can communicate by means of a mailbox:

```
/* Process ORION */

#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <iodef.h>

/* I/O status block */
struct {
                unsigned short iostat, iolen;
                unsigned int remainder;
}mbxiosb;

main() {
        void *p1, mbxast();
        char mbuffer[128], prmflg=0;
        unsigned short mbxchan, mbxiosb;
        unsigned int status, mbuflen=128, bufquo=384, promsk=0, outlen;
        $DESCRIPTOR(mblognam,"GROUP100_MAILBOX");

/* Create a mailbox */
        status = SYS$CREMBX( prmflg,         /* Permanent or temporary */  1
                             &mbxchan,       /* chan - channel number */
                             mbuflen,        /* maxmsg - buffer length */
                             bufquo,         /* bufquo - quota */
                             promsk,         /* promsk - protection mask */
                             0,              /* acmode - access mode */
                             &mblognam,      /* lognam - mailbox logical name */
                             0);             /* flags -  options */
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
/* Request I/O */
        status = SYS$QIO(0,               /* efn - event flag */   2
                          mbxchan,        /* chan - channel number */
                          IO$_READVBLK,   /* func - function modifier */
                          &mbxiosb,       /* iosb - I/O status block */
                          &mbxast,        /* astadr - AST routine */
                          &mbuffer,       /* p1 - output buffer */
                          mbuflen);       /* p2 - length of buffer */

        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
}
void mbxast(void) {                                                     3

        if (mbxiosb.iostat != SS$_NORMAL)

        status = SYS$QIOW( . . .  , &mbuffer, &outlen, . . .  )
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

        return;
}

/* Process Cygnus */

#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <iodef.h>
```

```
main() {

        unsigned short int mailchan;
        unsigned int status, outlen;
        char outbuf[128];
        $DESCRIPTOR(mailbox,"GROUP100_MAILBOX");

        status = SYS$ASSIGN(&mailbox, &mailchan, 0, 0, 0);           4
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        status = SYS$QIOW(0, mailchan, 0, 0, 0, 0, &outbuf, outlen, 0, 0, 0, 0)
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
}
```

1 Process ORION creates the mailbox and receives the channel number at MBXCHAN.

The **prmflg** argument indicates that the mailbox is a temporary mailbox. The logical name is entered in the LNM$TEMPORARY_MAILBOX logical name table.

The **maxmsg** argument limits the size of messages that the mailbox can receive. Note that the size indicated in this example is the same size as the buffer (MBUFFER) provided for the SYS$QIO request. A buffer for mailbox I/O must be at least as large as the size specified in the **maxmsg** argument.

When a process creates a temporary mailbox, the amount of system memory allocated for buffering messages is subtracted from the process's buffer quota. Use the **bufquo** argument to specify how much of the process quota should be used for mailbox message buffering.

Mailboxes are protected devices. By specifying a protection mask with the **promsk** argument, you can restrict access to the mailbox. (In this example, all bits in the mask are clear, indicating unlimited read and write access.)

2 After creating the mailbox, process ORION calls the SYS$QIO system service, requesting that it be notified when I/O completes (that is, when the mailbox receives a message) by means of an AST interrupt. The process can continue executing, but the AST service routine at MBXAST will interrupt and begin executing when a message is received.

3 When a message is sent to the mailbox (by CYGNUS), the AST is delivered and ORION responds to the message. Process ORION gets the length of the message from the first word of the I/O status block at MBXIOSB and places it in the longword OUTLEN so it can pass the length to SYS$QIOW_S.

4 Process CYGNUS assigns a channel to the mailbox, specifying the logical name the process ORION gave the mailbox. The SYS$QIOW system service writes a message from the output buffer provided at OUTBUF.

Note that on a write operation to a mailbox, the I/O is not complete until the message is read, unless you specify the IO$M_NOW function modifier. Therefore, if SYS$QIOW (without the IO$M_NOW function modifier) is used to write the message, the process will not continue executing until another process reads the message.

9.20.1 Mailbox Name

The **lognam** argument to the SYS$CREMBX service specifies a descriptor that points to a character string for the mailbox name.

Translation of the **lognam** argument proceeds as follows:

1. The current name string is prefixed with MBX$ and the result is subject to logical name translation.
2. If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM$C_MAXDEPTH.
3. The MBX$ prefix is stripped from the current name string that could not be translated. This current string is made a logical name with an equivalence name MBA*n* (*n* is a number assigned by the system).

For example, assume that you have made the following logical name assignment:

```
$ DEFINE MBX$CHKPNT CHKPNT_001
```

Assume also that your program contains the following statements:

```
        $DESCRIPTOR(mbxdesc,"CHKPNT");
   .
   .
   .
        status = SYS$CREMBX( . . .  ,&mbxdesc, . . .  );
```

The following logical name translation takes place:

1. MBX$ is prefixed to CHKPNT.
2. MBX$CHKPNT is translated to CHKPNT_001.

Because further translation is unsuccessful, the logical name CHKPNT_001 is created with the equivalence name MBA*n* (*n* is a number assigned by the system).

There are two exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), the operating system strips the underscore and considers the resultant string to be the actual name (that is, further translation is not performed).
- If the name string is the result of a logical name translation, then the name string is checked to see whether it has the **terminal** attribute. If the name string is marked with the **terminal** attribute, the operating system considers the resultant string to be the actual name (that is, further translation is not performed).

9.20.2 System Mailboxes

The system uses mailboxes for communication among system processes. All system mailbox messages contain, in the first word of the message, a constant that identifies the sender of the message. These constants have symbolic names (defined in the $MSGDEF macro) in the following format:

MSG$_sender

The symbolic names included in the $MSGDEF macro and their meanings are as follows:

Symbolic Name	Meaning
MSG$_TRMUNSOLIC	Unsolicited terminal data
MSG$_CRUNSOLIC	Unsolicited card reader data
MSG$_ABORT	Network partner aborted link
MSG$_CONFIRM	Network connect confirm
MSG$_CONNECT	Network inbound connect initiate
MSG$_DISCON	Network partner disconnected
MSG$_EXIT	Network partner exited prematurely
MSG$_INTMSG	Network interrupt message; unsolicited data
MSG$_PATHLOST	Network path lost to partner
MSG$_PROTOCOL	Network protocol error
MSG$_REJECT	Network connect reject
MSG$_THIRDPARTY	Network third-party disconnect
MSG$_TIMEOUT	Network connect timeout
MSG$_NETSHUT	Network shutting down
MSG$_NODEACC	Node has become accessible
MSG$_NODEINACC	Node has become inaccessible
MSG$_EVTAVL	Events available to DECnet Event Logger
MSG$_EVTRCVCHG	Event receiver database change
MSG$_INCDAT	Unsolicited incoming data available
MSG$_RESET	Request to reset the virtual circuit
MSG$_LINUP	PVC line up
MSG$_LINDWN	PVC line down
MSG$_EVTXMTCHG	Event transmitter database change

The remainder of the message contains variable information, depending on the system component that is sending the message.

The format of the variable information for each message type is documented with the system function that uses the mailbox.

9.20.3 Mailboxes for Process Termination Messages

When a process creates another process, it can specify the unit number of a mailbox as an argument to the Create Process ($CREPRC) system service. When you delete the created process, the system sends a message to the specified termination mailbox.

You cannot use a mailbox in memory shared by multiple processors as a process termination mailbox.

9.21 Example of Using I/O Services

In the following Fortran example, the first program, SEND.FOR, 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, RECEIVE.FOR, creates a mailbox with the same logical name, MAIL_BOX. It reads the messages from the mailbox into an array. It stops the read operations when a read operation generates an end-of-file message and the second longword of the I/O status block is nonzero. By checking that the I/O status block is nonzero, the second program confirms that the writing process sent the end-of-file message.

The processes use common event flag number 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 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
INTEGER*2 IOSTAT,
2         MSG_LEN
INTEGER READER_PID
COMMON /IOBLOCK/ IOSTAT,
2                MSG_LEN,
2                READER_PID

! 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$QIOW (,
2                    %VAL(MBX_CHAN),     ! Channel
2                    %VAL(WRITE_CODE),   ! I/O code
2                    IOSTAT,             ! Status block
2                    ,,
2                    %REF(MBX_MESSAGE),  ! P1
2                    %VAL(LEN),,,,)      ! P2
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
  IF (.NOT. IOSTAT) CALL LIB$SIGNAL (%VAL(STATUS))
END DO

! Write end of file
WRITE_CODE = IO$_WRITEOF .OR. IO$M_NOW
STATUS = SYS$QIOW (,
2                  %VAL(MBX_CHAN),     ! Channel
2                  %VAL(WRITE_CODE),   ! End of file code
2                  IOSTAT,             ! Status block
2                  ,,,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. IOSTAT) CALL LIB$SIGNAL (%VAL(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
INTEGER*2 IOSTAT,
2         MSG_LEN
INTEGER READER_PID
COMMON /IOBLOCK/ IOSTAT,
2                MSG_LEN,
2                READER_PID
```

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

! Read messages until cooperating process writes end-of-file
DO WHILE (.NOT. ((IOSTAT .EQ. SS$_ENDOFFILE) .AND.
2                (READER_PID .NE. 0)))

  STATUS = SYS$QIOW (,
2                    %VAL(MBX_CHAN),     ! Channel
2                    %VAL(READ_CODE),    ! Function code
2                    IOSTAT,             ! Status block
2                    ,,
2                    %REF(MBX_MESSAGE),  ! P1
2                    %VAL(LEN),,,,)      ! P2

   IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
   IF ((.NOT. IOSTAT) .AND.
2     (IOSTAT .NE. SS$_ENDOFFILE)) THEN
     CALL LIB$SIGNAL (%VAL(IOSTAT))
  ELSE IF (IOSTAT .NE. SS$_ENDOFFILE) THEN
    I = I + 1
    MESSAGES(I) = MBX_MESSAGE
    MESSAGE_LEN(I) = MSG_LEN
   END IF

END DO
                                .
                                .
                                .
```

10 Logical Name Services

This chapter describes how to create and use logical names. It contains the following sections:

Section 10.1 describes how to use logical name system services, how to use logical and equivalence names, and how to use logical name tables.

Section 10.2 describes how to create logical names.

Section 10.3 describes how to use SYS$CRELNT system service to create a logical name table.

Section 10.4 describes how to delete entries in a logical name table.

Section 10.5 describes how to translate a logical name to its equivalence string.

Section 10.6 shows a Fortran program that uses the logical name system services.

10.1 Logical Name System Services

This section describes how to use system services to establish logical names for general application purposes. The system performs special logical name translation procedures for names associated with I/O services and with services that can deal with facilities located in shared (multiport) memory. For further information, see the following chapters:

- Mailbox names and device names for I/O services: Chapter 9
- Common event flag cluster names: Chapter 14
- Global section names: Chapter 19

The operating system's logical name services provide a technique for manipulating and substituting character-string names. Logical names are commonly used to specify devices or files for input or output operations. You can use logical names to communicate information between processes by creating a logical name in one process in a shared logical name table and translating the logical name in another process. The operating system's logical name services are as follows:

- Create Logical Name (SYS$CRELNM)
- Create Logical Name Table (SYS$CRELNT)
- Delete Logical Name (SYS$DELLNM)
- Translate Logical Name (SYS$TRNLNM)

As the names of the logical name system services imply, when you use the logical name system services, you are concerned with creating, deleting, and translating logical names and with creating and deleting logical name tables.

The following sections describe various concepts you should be aware of when you use the logical name system services. For further discussion of logical names, see the *OpenVMS User's Manual*.

10.1.1 Logical Names, Equivalence Names, and Search Lists

A **logical name** is a user-specified character string that can represent a file specification, device name, logical name table name, application-specific information, or another logical name. Typically, for process-private purposes, you specify logical names that are easy to use and to remember. System managers and privileged users choose mnemonics for files, system devices, and search lists that are frequently accessed by all users.

An **equivalence string**, or an **equivalence name**, is a character string that denotes the actual file specification, device name, or character string. An equivalence name can also be a logical name. In this case, further translation is necessary to reveal the actual equivalence name, if permitted.

A multivalued logical name, commonly called a **search list**, is a logical name that has more than one equivalence string. Each equivalence string in the search list is assigned an index number starting at zero.

Logical names and their equivalence strings are stored in logical name tables. Logical names can have a maximum length of 255 characters. Equivalence strings can have a maximum of 255 characters. You can establish logical name and equivalence string pairs as follows:

- At the command level, with the DCL commands ALLOCATE, ASSIGN, DEFINE, or MOUNT
- In a program, with the Create Logical Name (SYS$CRELNM), Create Mailbox and Assign Channel (SYS$CREMBX), or Mount Volume (SYS$MOUNT) system service

For example, you could use the symbolic name TERMINAL to refer to an output terminal in a program. For a particular run of the program, you could use the DEFINE command to establish the equivalence name TTA2.

To perform an assignment in a program, you must define character-string descriptors for the name strings. In addition, you must call the system service through an external function declaration within your program, depending on the programming language.

10.1.2 Logical Name Tables

A logical name table contains logical name and equivalence string pairs. Each table is an independent name space. Logical name tables are referenced by logical names.

Logical name tables can be created in process space or in system space. Tables created in process space are accessible only by that process. Tables created in system space are potentially shareable among many processes. Certain logical name tables have predefined logical names that provide the environment for creating, deleting, and translating user-specified logical names. These predefined logical names begin with the prefix LNM$. Logical name and equivalence name pairs are maintained in three types of logical name tables:

- Directory tables
- Default tables
- User-defined name tables

When the process is created, the logical name directory tables and the default logical name tables are created for each new process.

10.1.2.1 Logical Name Directory Tables

Because the names of logical name tables are logical names, table names must reside in logical name tables. Two special tables called **directories** exist for this purpose. Table names are translated from these logical name directory tables. Logical name and equivalence name pairs for logical name tables are maintained in the following two directory tables:

- Process directory table (LNM$PROCESS_DIRECTORY)
- System directory table (LNM$SYSTEM_DIRECTORY)

The process directory table contains the names of all process-private user-defined logical name tables created through the SYS$CRELNT system service. In addition, the process directory table contains system-assigned logical name table names, the name of the process logical name table LNM$PROCESS_TABLE, and the default logical name table search list.

The system directory table contains the names of potentially shareable logical name tables and system-assigned logical name table names. You must have the SYSPRV privilege to create a logical name in the system directory table. For a discussion of privileges, see Section 10.1.4.

Logical names other than logical name table names can exist within these tables. The length of the logical names created in either of these tables must not exceed 31 characters. Logical names created in the directory tables must consist of alphanumeric characters, dollar signs ($), and underscores (_). Equivalence strings must not exceed 255 characters.

10.1.2.2 Process, Job, Group, and System Default Logical Name Tables

Certain logical name tables are created for or assigned to a process at process creation. These tables are called the **default logical name tables**. The newly created process is provided with these tables by default. Logical name and equivalence name pairs are maintained in the default logical name tables.

Each default logical name table has a logical name associated with it. To place an entry in a logical name table, specify a logical name table name. The default logical name table names and the common logical names used to refer to them are as follows:

Table	Name	Logical Name
Process	LNM$PROCESS_TABLE	LNM$PROCESS
Job	LNM$JOB_*xxxxxxxx*	LNM$JOB
Group	LNM$GROUP_*gggggg*	LNM$GROUP
System	LNM$SYSTEM_TABLE	LNM$SYSTEM

The letter *x* represents a numeral in an 8-digit hexadecimal number that uniquely identifies the job logical name table. The letter *g* represents a numeral in a 6-digit octal number that contains the user's group number.

The length of the logical names created in these tables must not exceed 255 characters, with no restriction on the types of characters used. Equivalence strings must not exceed 255 characters.

Process Logical Name Table

The process logical name table LNM$PROCESS_TABLE contains names used exclusively by the process. A process logical name table exists for each process in the system. Some entries in the process logical name table are made by system programs executing at more privileged access modes; these entries are qualified by the access mode from which the entry was made. The process logical name table contains the following process-permanent logical names:

Logical Name	Meaning
SYS$INPUT	Default input stream
SYS$OUTPUT	Default output stream
SYS$COMMAND	Original first-level (SYS$INPUT) input stream
SYS$ERROR	Default device to which the system writes error messages

SYS$COMMAND is created only for processes that execute LOGINOUT.

Process-Private Logical Name Creation and Image Rundown

Usually, you create logical names only in your process logical name table. Most entries in the process logical name table are made in user or supervisor mode. The following example shows how process-private logical names can be created in user mode by an image:

```
#include <stdio.h>
#include <lnmdef.h>
#include <ssdef.h>
#include <descrip.h>

/* Define an item  descriptor */
struct lst {
            unsigned short buflen, item_code;
            void *bufaddr;
            void *retlenaddr;
};

/* Declare an item list */
struct {
        struct lst items[2];
        unsigned int terminator;
}item_lst;

/* Equivalence name strings */

static char eqvnam1[] = "XYZ";
static char eqvnam2[] = "DEF";

main() {

        unsigned int status;
        $DESCRIPTOR(logdesc,"ABC");
        $DESCRIPTOR(tabdesc,"LNM$PROCESS");

        item_lst.items[0].buflen = strlen(eqvnam1);
        item_lst.items[0].item_code = LNM$_STRING;
        item_lst.items[0].bufaddr = eqvnam1;
        item_lst.items[0].retlenaddr = 0;

        item_lst.items[1].buflen = strlen(eqvnam2);
        item_lst.items[1].item_code = LNM$_STRING;
        item_lst.items[1].bufaddr = eqvnam2;
        item_lst.items[1].retlenaddr = 0;
        item_lst.terminator = 0;
/* Create a logical name */
        status = SYS$CRELNM( 0,   /* attr - attributes of logical name */
                  &tabdesc,       /* tabnam - name of logical name table */
                  &logdesc,       /* lognam - name of logical name */
                  0,              /* acmode - access mode 0 means use the */
                                  /* access mode of the caller=user mode */
                  &item_lst);     /* itm_lst - item list */
        if((status & 1) != 1)
                LIB$SIGNAL(status);

}
```

In the preceding example, logical name ABC was created and represents a search list with two equivalence strings, XYZ and DEF. Each time the LNM$_STRING item code of the **itmlst** argument is invoked, an index value is assigned to the next equivalence string. The newly created logical name and its equivalence string are contained in the process logical name table LNM$PROCESS_TABLE.

The following example illustrates the creation of a logical name in supervisor mode through DCL:

```
$ DEFINE/SUPERVISOR_MODE/TABLE=LNM$PROCESS ABC XYZ,DEF
```

In the preceeding example, supervisor mode and /TABLE=LNM$PROCESS are the defaults (default mode and default table) for the DEFINE command.

Process logical names that are created in user mode are deleted whenever the creating process runs an image down. The following DCL commands illustrate this behavior:

```
$ DEFINE/USER ABC XYZ
$ SHOW TRANSLATION ABC
  ABC   = XYZ
$ DIRECTORY
$ SHOW LOGICAL ABC
  ABC   = (undefined)
```

The DCL command DIRECTORY performs image rundown when it is finished operating. At that time, all user-mode process-private logical names are deleted, including the logical name ABC.

Job Logical Name Table

The job logical name table is a shareable table accessible by all processes within the same job tree. Whenever a detached process is created, a job logical name table is created for this process and all of its potential subprocesses. At the same time, the process-private logical name LNM$JOB is created in the process directory logical name table LNM$PROCESS_DIRECTORY. The logical name LNM$JOB translates to the name of the job logical name table.

Because the job logical name table already exists for the main process, only the process-private logical name LNM$JOB is created when a subprocess is created.

The job logical name table contains the following three process-permanent logical names for processes that execute LOGINOUT:

Logical Names	Meaning
SYS$LOGIN	Original default device and directory

Logical Names	Meaning
SYS$LOGIN_DEVICE	Original default device
SYS$SCRATCH	Default device and directory to which temporary files are written

Thus, instead of creating these logical names within the process logical name table LNM$PROCESS_TABLE for every process within a job tree, LOGINOUT creates these logical names once when it is executed for the process at the root of the job tree.

Additionally, the job logical name table contains the following logical names:

- The logical name optionally specified and associated with a newly created temporary mailbox
- The logical name optionally specified and associated with a privately mounted volume

You do not need special privileges to modify the job logical name table. For a discussion of privileges, see Section 10.1.4.

Group Logical Name Table

The group logical name table contains names that cooperating processes in the same group can use. You need the GRPNAM privilege to add or delete a logical name in the group logical name table. For a discussion of privileges, see Section 10.1.4.

Group logical name tables are created as needed. However, the logical name LNM$GROUP exists in each process's process directory LNM$PROCESS_ DIRECTORY. This logical name translates into the name of the group logical name table.

System Logical Name Table

The system logical name table LNM$SYSTEM_TABLE contains names that all processes in the system can access. This table includes the default names for all system-assigned logical names. You need the SYSNAM or SYSPRV privilege to add or delete a logical name in the system logical name table. For a discussion of privileges, see Section 10.1.4.

10.1.2.3 Creating User-Defined Logical Name Tables

You can create process-private tables and shareable tables by calling the SYS$CRELNT system service in a program. However, you must have SYSPRV privilege to create a shareable table. For a discussion of privileges, see Section 10.1.4. Processes other than the creating process cannot use logical names contained in process-private tables.

Logical name tables are created through the SYS$CRELNT system service either with the DCL command CREATE/NAME_TABLE or by calling SYS$CRELNT in a program. If granted access, processes other than the creating process can use shareable tables.

The length of logical names created in user-defined logical name tables must not exceed 255 characters. Equivalence strings must not exceed 255 characters.

10.1.3 Duplicating Logical Names

A logical name table can contain entries for the same logical name at different access modes. Different logical name tables can contain entries for the same logical name. In all other cases, only one entry can exist for a particular logical name in a logical name table.

Because identical logical names can exist in more than one logical name table, the logical name that the system uses depends on the order in which it searches the logical name tables. For example, when the system attempts to translate a logical name to identify the location of a file, it uses the logical name LNM$FILE_DEV to provide the list of tables in which to look for the name. The order in which the tables are listed is also the order in which they are searched.

By default, the precedence order defined by LNM$FILE_DEV is:

1. Process table
2. Job table
3. Group table
4. System table

If, for example, a logical name exists in both the process and the group logical name tables, the logical name within the process table is used.

By default, the DEFINE and DEASSIGN commands place names in, and delete names from, your process table. However, you can request a different table with the /TABLE qualifier, as shown in the following example:

```
$ DEFINE/TABLE=LNM$SYSTEM REVIEWERS DISK3:[PUBLIC]REVIEWERS.DIS
```

Any number of logical names can have the same equivalence name. Consider the following examples of the logical name TERMINAL defined in several tables. The logical name TERMINAL translates differently depending on the table specified.

Process Logical Name Table for Process A

The following process logical name table equates the logical name TERMINAL to the specific terminal TTA2. The INFILE and OUTFILE logical names are equated to disk specifications. The logical names were created from supervisor mode.

Logical Name	Equivalence Name	Access Mode
INFILE	DM1:[HIGGINS]TEST.DAT	Supervisor
OUTFILE	DM1:[HIGGINS]TEST.OUT	Supervisor
TERMINAL	TTA2:	Supervisor
.	.	.
.	.	.
.	.	.

To determine the equivalence string for the logical name TERMINAL in the preceding table, enter the following command:

```
$ SHOW LOGICAL TERMINAL
```

The system returns the equivalence string TTA2:.

Job Logical Name Table

The portion of the following job logical name table assigns the logical name TERMINAL to a virtual terminal VTA14. The logical name SYS$LOGIN is the device and directory for the process when you log in. The SYS$LOGIN logical name is defined in executive mode.

Logical Name	Equivalence Name	Access Mode
SYS$LOGIN	DBA9:[HIGGINS]	Executive
TERMINAL	VTA14:	User
.	.	.
.	.	.
.	.	.

To determine the equivalence string of the logical name TERMINAL defined in the preceding table, enter the following command:

```
$ SHOW LOGICAL/JOB TERMINAL
```

The system returns the equivalence string VTA14: as the translation.

User-Defined Logical Name Table

The following user-defined logical name table (called LOG_TBL for purposes of this discussion) contains a definition of TERMINAL as the mailbox device MBA407. The multivalued logical name (search list) XYZ has two translations: DISK1 and DISK3.

Logical Name	Equivalence Name	Access Mode
TERMINAL	MBA407:	Supervisor
XYZ	DISK1:	Supervisor
	DISK3:	
.	.	.
.	.	.
.	.	.

To determine the equivalence string for the logical name TERMINAL in the preceding user-defined table, enter the following command:

```
$ SHOW LOGICAL/TABLE=LOG_TBL TERMINAL
```

The system returns the equivalence string MBA407. In order to use this definition of TERMINAL as a device or file specification, you must redefine the logical name LNM$FILE_DEV to reference the user-defined table, as follows:

```
$ DEFINE/TABLE=LNM$PROCESS_DIRECTORY LNM$FILE_DEV LOG_TBL, -
_$ LNM$PROCESS_TABLE,LNM$JOB,LNM$GROUP,LNM$SYSTEM_TABLE
```

In this example, the DCL command DEFINE is used to redefine the default search list LNM$FILE_DEV. The /TABLE qualifier specifies the table LNM$PROCESS_DIRECTORY that is to contain the redefined search list. The system searches the tables defined by LNM$FILE_DEV in the following order: LOG_TBL, LNM$PROCESS_TABLE, LNM$JOB, LNM$GROUP, and LNM$SYSTEM_TABLE.

System Logical Name Table

The following system logical table contains system-assigned logical names accessible to all processes in the system. For example, the logical names SYS$LIBRARY and SYS$SYSTEM provide logical names that all users can access to use the device and directory that contain system files.

Logical Name	Equivalence Name
SYS$LIBRARY	SYS$SYSROOT:[SYSLIB]
SYS$SYSTEM	SYS$SYSROOT:[SYSEXE]
.	.
.	.
.	.

The Logical Names section of the *OpenVMS User's Manual* contains a list of these system-assigned logical names.

Logical Name Supersession

If the logical name TERMINAL is equated to TTA2 in the process table, as shown in the previous examples, and the process subsequently equates the logical name TERMINAL to TTA3, the equivalence of TERMINAL TTA2 is replaced by the new equivalence name. The successful return status code SS$_SUPERSEDE indicates that a new entry replaced an old one.

The definitions of TERMINAL in the job table and in the user-defined table LOG_TBL are unaffected.

10.1.4 Defining Privileges

Certain functions of the logical name system services are restricted to users with specific privileges. The system checks the privileges in the user authorization file (UAF) granted to you when your system manager sets up your account. The system also checks for read, write, and delete accessibility. Privileges allow users to perform the functions shown in Table 10–1.

Table 10–1 Summary of Privileges

Privilege	Function
GRPNAM	Creates or deletes a logical name in your group logical name table.
GRPPRV	Creates or deletes a logical name in your group logical name table.
SYSNAM	Creates executive- or kernel-mode logical names. Deletes a logical name or table at an inner access mode.
SYSPRV	Creates or deletes a logical name in your group logical name table. Creates a shareable table.

All users can create, delete, and translate their own process-private logical names and process-private logical name tables.

10.1.5 Specifying Access Modes

You can specify the access mode of a logical name when you define the logical name. If you do not specify an access mode, then the access mode defaults to that of the caller of the SYS$CRELNM system service. If you specify the **acmode** argument and the process has SYSNAM privilege, the logical name is created with the specified access mode. Otherwise, the access mode can be no more privileged than the caller. For information about access modes, see *OpenVMS Programming Interfaces: Calling a System Routine* and the discussion of SYS$CRELNM in the *OpenVMS System Services Reference Manual*.

A logical name table can contain multiple definitions of the same logical name with different access modes. If a request to translate such a logical name specifies the **acmode** argument, then the SYS$TRNLNM system service ignores all names defined at a less privileged mode. A request to delete a logical name includes the access mode of the logical name. Unless the process has the SYSNAM privilege, the mode specified can be no more privileged than the caller.

The command interpreter places entries made from the command stream into the process-private logical name table; these are supervisor-mode entries and are not deleted at image exit (except for the logical names defined by the DCL commands ASSIGN/USER and DEFINE/USER). During certain system operations, such as the activation of an image installed with privilege, only executive- and kernel-mode logical names are used.

Logical names or logical name table names, which either an image running in user mode or the DCL commands ASSIGN/USER and DEFINE/USER have placed in a process-private logical name table, are automatically deleted at image exit. Shareable user-mode names, however, survive image exit and process deletion.

10.1.6 Specifying Attributes

Generally, attributes specified through the logical name system services perform two functions: they affect the creation of logical names or govern how the system service operates, and they affect the translation of logical names and equivalence strings.

Attributes that affect the creation of the logical names are specified optionally in the **attr** argument of a system service call.

You can specify any of the following attributes:

- LNM$M_CONCEALED—Specifies that the equivalence string for the logical name is an OpenVMS RMS concealed device name.
- LNM$M_CONFINE—Prevents process-private logical names from being copied to subprocesses. Subprocesses are created by the DCL command SPAWN or by the run-time library LIB$SPAWN routine. This attribute is specified only in a SYS$CRELNM or SYS$CRELNT system service call.
- LNM$M_NO_ALIAS—Prevents creation of a duplicate logical name in the specified logical name table at an outer access mode. If another logical name already exists in the table at an outer access mode, that name is deleted.

  If specified in a SYS$CRELNT system service call, this attribute prevents creation of a logical name table at an outer access mode in a directory table if the table name already exists in the directory table.

  This attribute is specified only in a SYS$CRELNM or SYS$CRELNT system service call.

- LNM$M_CREATE_IF—Prevents creation of a logical name table if the specified table already exists at the specified access mode in the appropriate directory table. This attribute is specified only in a SYS$CRELNT system service call.
- LNM$M_CASE_BLIND—Governs the translation process and causes SYS$TRNLNM to ignore uppercase and lowercase differences in letters when searching for logical names. This attribute is specified only in a SYS$TRNLNM system service call.
- LNM$M_TERMINAL—Prevents further translation of equivalence strings by the logical name services.

The translation attributes LNM$M_CONCEALED and LNM$M_TERMINAL associated with logical names and equivalence strings are specified optionally through the LNM$_ATTRIBUTES item code in the **itmlst** argument of the SYS$CRELNM system service call. When the item code LNM$_ATTRIBUTES is specified through SYS$TRNLNM, the system returns the current attributes associated with the logical name and equivalence string at the current index value. Since a logical name can have more than one equivalence name, each equivalence name is identified by an index value. The item code LNM$_INDEX of SYS$TRNLNM searches for an equivalence name that has the specified index value.

The following attributes may be returned:

- LNM$M_CONCEALED—Indicates that the equivalence string at the current index value for the logical name is an OpenVMS RMS concealed device name.
- LNM$M_CONFINE—Indicates that the logical name cannot be used by spawned subprocesses. Subprocesses are created by the DCL command SPAWN or by the run-time library LIB$SPAWN routine.
- LNM$M_CRELOG—Indicates that the logical name was created by the SYS$CRELOG system service.
- LNM$M_EXISTS—Indicates that the equivalence string at the specified index value exists.
- LNM$M_NO_ALIAS—Indicates that if the logical name already exists in the table, it cannot be created in that table at an outer access mode.
- LNM$M_TABLE—Indicates that the logical name is the name of a logical name table.
- LNM$M_TERMINAL—Indicates that the equivalence strings cannot be translated further.

The attributes of multiple equivalence strings do not have to match. For more information about attributes, refer to the appropriate system service in the *OpenVMS System Services Reference Manual*.

10.1.7 Establishing Logical Name Table Quotas

A logical name table **quota** is the number of bytes allocated in memory for logical names contained in a logical name table. Logical name table quotas are established in the following instances:

- When the system is initialized
- When a process is created

- When logical name tables are created

Each logical name table has a quota associated with it that limits the number of bytes of memory (either process pool or system paged pool) that can be occupied by the names defined in the table. The quota for a table is established when the table is created.

If no quota is specified, the newly created table has unlimited quota. Note that this table can expand to consume all available process or system memory, and all users with write access to such a shareable table can cause the unlimited consumption of system paged pool.

10.1.7.1 Directory Table Quotas

When the system is initialized, unlimited quota is automatically established for the system directory table LNM$SYSTEM_DIRECTORY.

When you log in to the system, unlimited quota is automatically established for the process directory table LNM$PROCESS_DIRECTORY.

10.1.7.2 Default Logical Name Table Quotas

The process, group, and system logical name tables have unlimited quota.

10.1.7.3 Job Logical Name Table Quotas

Because the job logical name table is a shareable table, and because you do not need special privileges to create logical names within it, the quota allocated to this logical name table is constrained at the time the table is created. The following three mechanisms exist to specify the quota for the job logical name table at the time of its creation:

- For all processes that activate LOGINOUT, the quota for the job logical name table is obtained from the system authorization file. This allows the quota for the job to be specified on a user-by-user basis. You can modify the job logical name table quota by specifying a value with the DCL command AUTHORIZE /JTQUOTA.
- For all processes that do not activate LOGINOUT, the quota for the job logical name table can be specified as a quota list item (PQL$_JTQUOTA) in the call to the Create Process (SYS$CREPRC) system service. If a detached process is to be created by means of the DCL command RUN/DETACHED, then you can use the /JOB_TABLE_QUOTA qualifier to specify the SYS$CREPRC quota list item.
- For all processes that do not activate LOGINOUT and do not specify a PQL$_ JTQUOTA quota list item in their call to SYS$CREPRC, the quota for the job logical name table is taken from the dynamic System Generation utility (SYSGEN) parameter PQL$_DJTQUOTA. You can use SYSGEN to display both PQL$_DJTQUOTA and PQL$_MJTQUOTA, the default and minimum job logical name table quotas, respectively.

10.1.7.4 User-Defined Logical Name Table Quotas

User-defined logical name tables can be created with either an explicit limited quota or no quota limit.

The presence of user-defined logical name table quotas eliminates the need for a privilege (for example, SYSNAM or GRPNAM) to control consumption of paged pool when you create logical names in a shareable table.

10.1.8 Using Logical Name and Equivalence Name Format Conventions

The operating system uses special conventions for assigning logical names to equivalence names and for translating logical names. These conventions are generally transparent to user programs; however, you should be aware of the programming considerations involved.

If a logical name string presented in I/O services is preceded by an underscore (_), the I/O services bypass logical name translation, drop the underscore, and treat the logical name as a physical device name.

When you log in, the system creates default logical name table entries for process-permanent files. The equivalence names for these entries (for example, SYS$INPUT and SYS$OUTPUT) are preceded by a 4-byte header that contains the following information:

Byte	Contents
0	^X1B (escape character)
1	^X00
2–3	OpenVMS RMS Internal File Identifier (IFI)

This header is followed by the equivalence name string. If any of your program applications must translate system-assigned logical names, you must prepare the program to check for the existence of this header and to use only the desired part of the equivalence string. The following program demonstrates how to do this:

```
#include <stdio.h>
#include <lnmdef.h>
#include <ssdef.h>
#include <descrip.h>
#include <ctype.h>
#include <string.h>

#define HEADER 4

/* Define an item descriptor */
struct {
        unsigned short buflen, item_code;
        void *bufaddr;
        void *retlenaddr;
        unsigned int terminator;
}item_lst;


main() {

        unsigned int status,len,i;
        char resstring[LNM$C_NAMLENGTH];
        $DESCRIPTOR(tabdesc,"LNM$FILE_DEV");
        $DESCRIPTOR(logdesc,"SYS$OUTPUT");

        item_lst.buflen = LNM$C_NAMLENGTH;
        item_lst.item_code = LNM$_STRING;
        item_lst.bufaddr = resstring;
        item_lst.retlenaddr = 0;
        item_lst.terminator = 0;
/* Translate the logical name */
        status = SYS$TRNLNM( 0,   /* attr - attributes of the logical name */
                  &tabdesc,       /* tabnam - logical name table */
                  &logdesc,       /* lognam - logical name */
                  0,              /* acmode - accessm mode */
                  &item_lst);     /* itmlst - item list */
        if((status & 1) != 1)
                LIB$SIGNAL( status );

/*
   Examine 4-byte header
   Is first character an escape char?
   If so, dump the header
*/
        if( resstring[0] == 0x1B) {
                printf("\nDumping the header...\n");
                for(i = 0; i < HEADER; i++)
                        printf(" Byte %d: %X\n",i,resstring[i]);

                printf("\nEquivalence string: %s\n",(resstring + HEADER));
        }
        else
                printf("Header not found\n");

}
```

10.1.9 Specifying the Logical Name Table Search List

Logical names exist as entries within logical name tables. When a logical name is to be created, deleted, or translated, you must specify or take the default name that designates the logical name table that contains the logical name. This name possesses one or more of the following characteristics:

- It is the name of a logical name table.
- It is a logical name that iteratively translates in the process or system directory table to the name of a logical name table.
- It is a multivalued logical name (search list) that iteratively translates to the names of several logical name tables. The tables are used in the order in which they appear.

As mentioned in Section 10.1.2, predefined logical names exist for certain logical name tables. These predefined names begin with the prefix LNM$. You can redefine these names to modify the search order or the tables used.

Instead of a fixed set of logical name tables and a rigidly defined order (process, job, group, system) for searching those tables, you can specify which tables are to be searched and the order in which they are to be searched. Logical names in the directory tables are used to specify this searching order. By convention, each class of logical name (for example, device or file specification) uses a particular predefined name for this purpose.

For example, LNM$FILE_DEV is the logical name that defines the list of logical name tables used whenever file specifications or device names are translated by OpenVMS RMS or the I/O services. LNM$FILE_DEV is the default for file specifications and device names. This name must translate to a list of one or more logical name table names that specify the tables to be searched when translating file specifications.

By default, LNM$FILE_DEV specifies that the process, job, group, and system tables are all searched, in that order, and that the first match found is returned.

Logical name table names are translated from two tables: the process logical name directory table LNM$PROCESS_DIRECTORY and the system logical name directory table LNM$SYSTEM_DIRECTORY. The LNM$FILE_DEV logical name table must be defined in one of these tables.

Thus, if identical logical names exist in the process and group tables, the process table entry is found first, and the job and group tables are not searched. When the process logical name table is searched, the entries are searched in order of access mode, with user-mode entries matched first, supervisor-mode entries second, and so on.

If you want to change the list of tables used for device and file specifications, you can redefine LNM$FILE_DEV in the process directory table LNM$PROCESS_DIRECTORY.

10.2 Creating a Logical Name Using SYS$CRELNM

To perform an assignment in a program, you must provide character-string descriptors for the name strings, select the table to contain the logical name, and use the SYS$CRELNM system service as shown in the following example. In either case, the result is the same: the logical name DISK is equated to the physical device name DUA2 in table LNM$JOB.

```
#include <stdio.h>
#include <lnmdef.h>
#include <descrip.h>
#include <string.h>
#include <ssdef.h>

/* Define an item descriptor */

struct itm {
            unsigned short buflen, item_code;
            void *bufaddr;
            void *retlenaddr;
};

/* Declare an item list */

struct {
                struct itm items[2];
                unsigned int terminator;
}itm_lst;

main() {

        static char eqvnam[] = "DUA2:";
        unsigned int status, lnmattr;
        $DESCRIPTOR(logdesc,"DISK");
        $DESCRIPTOR(tabdesc,"LNM$JOB");

        lnmattr = LNM$M_TERMINAL;

/* Initialize the item list */

        itm_lst.items[0].buflen = 4;
        itm_lst.items[0].item_code = LNM$_ATTRIBUTES;
        itm_lst.items[0].bufaddr = &lnmattr;
        itm_lst.items[0].retlenaddr = 0;

        itm_lst.items[1].buflen = strlen(eqvnam);
        itm_lst.items[1].item_code = LNM$_STRING;
        itm_lst.items[1].bufaddr = eqvnam;
        itm_lst.items[1].retlenaddr = 0;
        itm_lst.terminator = 0;
/* Create the logical name */
        status = SYS$CRELNM(0,           /* attr - attributes */
                        &tabdesc,       /* tabnam - logical table name */
                        &logdesc,       /* lognam - logical name */
                        0,              /* acmode - access mode */
                        &itm_lst);      /* itmlst - item list */
        if((status & 1) != 1)
                LIB$SIGNAL(status);
}
```

Note that the translation attribute is specified as terminal. This attribute indicates that iterative translation of the logical name DISK ends when the equivalence string DUA2 is returned. In addition, because the **acmode** argument was not specified, the access mode of the logical name DISK is the access mode of the calling image.

10.3 Creating Logical Name Tables Using SYS$CRELNT

The Create Logical Name Table (SYS$CRELNT) system service creates logical name tables. Logical name tables can be created in any access mode depending on the privileges of the calling process. A user-specified logical name that identifies the newly created logical name table is stored in the process directory table LNM$PROCESS_DIRECTORY.

The following example illustrates a call to the SYS$CRELNT system service:

```
#include <stdio.h>
#include <ssdef.h>
#include <lnmdef.h>
#include <descrip.h>

main() {

        unsigned int status, tab_attr=LNM$M_CONFINE, tab_quota=5000;
        $DESCRIPTOR(tabdesc,"LOG_TABLE");
        $DESCRIPTOR(pardesc,"LNM$PROCESS_TABLE");

/* Create the logical name table */
        status = SYS$CRELNT(&tab_attr,  /* attr - table attributes */
                0,                      /* resnam - logical table name */
                0,                      /* reslen - length of table name */
                &tab_quota,             /* quota - max no. of bytes allocated */
                                        /* for names in this table */
                0,                      /* promsk - protection mask */
                &tabdesc,               /* tabnam - name of new table */
                &pardesc,               /* partab - name of parent table */
                0);                     /* acmode - access mode */
        if((status & 1) != 1) {
                LIB$SIGNAL(status);
}
```

In this example, a user-defined table LOG_TABLE is created with an explicit quota of 5000 bytes. The name of the newly created table is an entry in the process-private directory LNM$PROCESS_DIRECTORY. The quota of 5000 bytes is deducted from the parent table LNM$PROCESS_TABLE. Because the CONFINE attribute is associated with the logical name table, the table cannot be copied from the process to its spawned processes.

10.3.1 Creating Shareable Logical Name Tables

If you have SYSPRV privilege, you can create shareable logical name tables. You can assign protection to these tables through the **promsk** argument of the SYS$CRELNT system service. The **promsk** argument allows you to specify the type of access for system, owner, group, and world users, as follows:

- Read privileges allow access to names in the logical name table.
- Write privileges allow creation and deletion of names within the logical name table.
- Delete privileges allow deletion of the logical name table.

Note

The E protection bit is reserved by Digital Equipment Corporation.

If the **promsk** argument is omitted, complete access is granted to system and owner, and no access is granted to group and world.

10.4 Deleting Logical Names Using SYS$DELLNM

The Delete Logical Name (SYS$DELLNM) system service deletes entries from a logical name table. When you write a call to the SYS$DELLNM system service, you can specify a single logical name to delete, or you can specify that you want to delete all logical names from a particular table. For example, the following call deletes the process logical name TERMINAL from the job logical name table:

```
#include <stdio.h>
#include <lnmdef.h>
#include <ssdef.h>
#include <descrip.h>

main() {

        unsigned int status;
        $DESCRIPTOR(logdesc,"DISK");
        $DESCRIPTOR(tabdesc,"LNM$JOB");

/* Delete the logical name */
        status = SYS$DELLNM(&tabdesc,   /* tabnam - logical table name */
                        &logdesc,       /* lognam - logical name */
                0);                     /* acmode - access mode */
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

}
```

For information about access modes and the deletion of logical names, see *OpenVMS Programming Interfaces: Calling a System Routine*.

10.5 Translating Logical Names Using SYS$TRNLNM

The Translate Logical Name (SYS$TRNLNM) system service translates a logical name to its equivalence string. In addition, SYS$TRNLNM returns information about the logical name and equivalence string.

The system service call to SYS$TRNLNM specifies the tables to search for the logical name. The **tabnam** argument can be either the name of a logical name table or a logical name that translates to a list of one or more logical name tables.

Because logical names can have many equivalence strings, you can specify which equivalence string you want to receive.

A number of system services that require a device name accept a logical name and translate the logical name iteratively until a physical device name is found (or until the system default number of logical name translations has been performed, typically 10). These services implicitly specify the logical name table name LNM$FILE_DEV. For more information about LNM$FILE_DEV, refer to Section 10.1.9.

The following system services perform iterative logical name translation automatically:

- Allocate Device (SYS$ALLOC)
- Assign I/O Channel (SYS$ASSIGN)
- Broadcast (SYS$BRDCST)
- Create Mailbox (SYS$CREMBX)
- Deallocate Device (SYS$DALLOC)
- Dismount Volume (SYS$DISMOU)
- Get Device/Volume Information (SYS$GETDVI)
- Mount Volume (SYS$MOUNT)

In many cases, however, a program must perform the logical name translation to obtain the equivalence name for a logical name outside the context of a device name or file specification. In that case, you must supply the name of the table or tables to be searched. The SYS$TRNLNM system service searches the user-specified logical name tables for a specified logical name and returns the equivalence name. In addition, SYS$TRNLNM returns attributes that are specified optionally for the logical name and equivalence string.

The following example shows a call to the SYS$TRNLNM system service to translate the logical name ABC:

```
#include <stdio.h>
#include <lnmdef.h>
#include <descrip.h>
#include <ssdef.h>

/* Define an item descriptor */

struct itm {
             unsigned short buflen, item_code;
             void *bufaddr;
             void *retlenaddr;
};

/* Declare an item list */
struct {
         struct itm items[2];
         unsigned int terminator;
}trnlst;

main() {

        char eqvbuf1[LNM$C_NAMLENGTH], eqvbuf2[LNM$C_NAMLENGTH];
        unsigned int status, trnattr=LNM$M_CASE_BLIND;
        unsigned int eqvdesc1, eqvdesc2;
        $DESCRIPTOR(logdesc,"ABC");
        $DESCRIPTOR(tabdesc,"LNM$FILE_DEV");
/* Assign values to the item list */

        trnlst.items[0].buflen = LNM$C_NAMLENGTH;
        trnlst.items[0].item_code = LNM$_STRING;
        trnlst.items[0].bufaddr = eqvbuf1;
        trnlst.items[0].retlenaddr = &eqvdesc1;

        trnlst.items[1].buflen = LNM$C_NAMLENGTH;
        trnlst.items[1].item_code = LNM$_STRING;
        trnlst.items[1].bufaddr = eqvbuf2;
        trnlst.items[1].retlenaddr = &eqvdesc2;
        trnlst.terminator = 0;

/* Translate the logical name */
        status = SYS$TRNLNM(&trnattr,  /* attr - attributes */
                       &tabdesc,       /* tabnam - table name */
                       &logdesc,       /* lognam - logical name */
                       0,              /* acmode - access mode */
                       &trnlst);       /* itmlst - item list */
        if((status & 1) != 1)
                LIB$SIGNAL(status);

}
```

This call to the SYS$TRNLNM system service results in the translation of the logical name ABC. In addition, LNM$FILE_DEV is specified in the **tabnam** argument as the search list that SYS$TRNLNM is to use to find the logical name ABC. The logical name ABC was assigned two equivalence strings. The LNM$_STRING item code in the **itmlst** argument directs SYS$TRNLNM to look for an equivalence string at the current index value. Note that the LNM$_STRING item code is invoked twice. The equivalence strings are placed in the two output buffers, EQVBUF1 and EQVBUF2, described by TRNLIST.

The attribute LNM$M_CASE_BLIND governs the translation process. The SYS$TRNLNM system service searches for the equivalence strings without regard to uppercase or lowercase letters. The SYS$TRNLNM system service matches any of the following character strings: ABC, aBC, AbC, abc, and so forth.

The output equivalence name string length is written into the first word of the character string descriptor. This descriptor can then be used as input to another system service.

10.6 Example of Using the Logical Name System Services

In the following example, the Fortran program CALC.FOR creates a spawned subprocess to perform an iterative calculation. The logical name REP_NUMBER specifies the number of times that REPEAT should perform the calculation. Because the two processes are part of the same job, REP_NUMBER is placed in the job logical name table LNM$JOB. (Note that logical name table names are case sensitive. Specifically, LNM$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

      INCLUDE '($LNMDEF)'
      INCLUDE '($SYSSRVNAM)'
      INTEGER*4 STATUS
```

```
 INTEGER*2 NAME_LEN,
2         NAME_CODE
 INTEGER*4 NAME_ADDR,
2         RET_ADDR /0/,
2         END_LIST /0/

 COMMON /LIST/ NAME_LEN,
2             NAME_CODE,
2             NAME_ADDR,
2             RET_ADDR,
2             END_LIST

 CHARACTER*3 REPETITIONS_STR
 INTEGER REPETITIONS

 EXTERNAL CLI$M_NOLOGNAM,
2        CLI$M_NOCLISYM,
2        CLI$M_NOKEYPAD,
2        CLI$M_NOWAIT

  NAME_LEN = 3
  NAME_CODE = (LNM$_STRING)
  NAME_ADDR = %LOC(REPETITIONS_STR)
  STATUS = SYS$CRELNM (,'LNM$JOB','REP_NUMBER',,NAME_LEN)
  IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

 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))

  END

 PROGRAM REPEAT
 INTEGER STATUS,
2       SYS$TRNLNM,SYS$DELLNM
 INTEGER*4   REITERATE,
2           REPEAT_STR_LEN
 CHARACTER*3 REPEAT_STR
! Item list for SYS$TRNLNM
 INTEGER*2 NAME_LEN,
2         NAME_CODE
 INTEGER*4 NAME_ADDR,
2         RET_ADDR,
2         END_LIST /0/
 COMMON /LIST/ NAME_LEN,
2             NAME_CODE,
2             NAME_ADDR,
2             RET_ADDR,
2             END_LIST

 NAME_LEN = 3
 NAME_CODE = (LNM$_STRING)
 NAME_ADDR = %LOC(REPEAT_STR)
 RET_ADDR = %LOC(REPEAT_STR_LEN)
 STATUS = SYS$TRNLNM (,
2                    'LNM$JOB',      ! Logical name table
2                    'REP_NUMBER',, ! Logical name
2                    NAME_LEN)      ! List requesting equivalence string
 IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

 READ (UNIT = REPEAT_STR,
2     FMT = '(I3)') REITERATE
```

```
 DO I = 1, REITERATE
 END DO

 STATUS = SYS$DELLNM ('LNM$JOB',     ! Logical name table
2                    'REP_NUMBER',) ! Logical name
 IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))

  END
```

11
Distributed Name Service (VAX Only)

This chapter describes the Digital Distributed Name Service (DECdns) Clerk by introducing the functions of the DECdns (SYS$DNS) system service and various run-time library routines. It is divided into the following sections:

Section 11.1 describes how to use the portable application programming interface and the operating system's system service and run-time library interface.

Section 11.2 describes how to use the SYS$DNS system service.

Section 11.3 describes how to use the DCL command DEFINE.

11.1 DECdns Clerk System Service

The DECdns Clerk (SYS$DNS) system service provides applications with a means of assigning networkwide names to system resources. Applications can use DECdns to name such resources as printers, files, disks, nodes, servers, and application databases. Once an application has named a resource using DECdns, the name is available for all users of the application.

The SYS$DNS system service supports two programming interfaces:

- Portable application programming interface
- System service and run-time library (RTL)

Portable Application Interface

Application designers should select an interface for their application based on programming language, application base, and specific requirements of their application.

The portable interface provides support for applications written in the C programming language, and it provides a high-level interface with easy-to-use methods of creating and maintaining DECdns names. Use the portable interface for applications that must be portable between VAX systems and the Digital UNIX operating system.

The portable interface is documented in the *Guide to Programming with DECdns*.

VAX System Services and RTL Routines

The VAX system services and run-time library routines can be used by applications written in the high-level and midlevel languages listed in the preface of this document. However, applications that use these interfaces are limited to the VAX system environment. Use the system service when an application meets any of the following requirements:

- The application needs the full capabilities, flexibility, and functions of asynchronous support.
- The application will run as part of a privileged shareable image on the operating system.

- The application is not written in the C programming language.

The SYS$DNS system service is documented in the *OpenVMS System Services Reference Manual*. Before using this system service, familiarize yourself with the basic operating principles, terms, and definitions used by DECdns. You can gain a working knowledge of DECdns by reading about the following topics in the *Guide to Programming with DECdns*:

- DECdns component operation
- Namespace directories, objects, soft links, groups, and clearinghouses
- DECdns name syntax
- Attributes
- Clerk caching
- Setting confidence and timeouts
- Recommendations for DECdns application programmers

By understanding these topics, you can proceed more easily with this chapter, which provides an introduction to the DECdns system service and run-time library routines and discusses the following topics:

- Functions provided by the service and routines
- How to use the SYS$DNS system service

11.1.1 Using the DECdns System Service and Run-Time Library Routines

You can use the SYS$DNS system service and run-time library routines together to assign, maintain, and retrieve DECdns names. This section describes the capabilities of each interface.

11.1.1.1 Using the SYS$DNS System Service

DECdns provides a single system service call (SYS$DNS) to create, delete, modify, and retrieve DECdns names from a namespace. The SYS$DNS system service completes asynchronously; that is, it returns to the client immediately after making a name service call. The status returned to the client indicates whether a request was queued successfully to the name service.

The SYS$DNSW system service is the synchronous equivalent of SYS$DNS. The SYS$DNSW call is identical to SYS$DNS in every way except that SYS$DNSW returns to the caller after the operation completes.

The SYS$DNS call has two main parameters:

- A function code that identifies the particular service to perform
- An item list that specifies all the parameters for the required function

The system service provides the following functions:

- Create and delete DECdns names in the namespace
- Enumerate DECdns names in a particular directory
- Add, read, remove, and test attributes and attribute values
- Add, create, remove, restore, and update directories
- Create, remove, and resolve soft links
- Create and remove groups

- Add, remove, and test members in a group
- Parse names to convert string format names to DECdns opaque format names and back to string

You specify item codes as either input or output parameters in the item list. Input parameters modify functions, set context, or describe the information to be returned. Output parameters return the requested information.

You can specify the following in input item codes:

- An attribute name and type
- The class of a DECdns name and, optionally, a class filter
- The class version of a DECdns name
- A confidence setting to indicate whether the request should be serviced from the clerk's cache or from a server
- An indication that the application will repeat a read call, which forces caching of recently read data
- A name or timestamp that sets the context from which to begin or restart enumerating or reading
- The name and type of an object, directory, group, member, clearinghouse, or soft link, and the ability to suppress the namespace nickname from the full name
- A simple or full name in opaque or string format
- A request to search subgroups for a member
- An operation, either adding or deleting an attribute
- A value for an attribute
- A pointer to the address of the next character in a full or simple name
- A timeout period to wait for a call to complete
- An expiration time and extension time for soft links

The output item codes return the following information:

- A creation timestamp for an object
- A set of child directories, soft links, attribute names, attribute values, or object names
- An opaque simple or full name
- A string name and length
- A resolved soft link
- A name or timestamp context variable that indicates the last directory, object, soft link, or attribute that was enumerated or read

11.1.1.2 Using the Run-Time Library Routines

You can use the DECdns run-time library routines to manipulate output from the SYS$DNS system service. The routines provide the following functions:

- Remove a value from a set returned by an enumeration or read system service function
- Compare, append, concatenate, and count opaque names that were created with the system service
- Convert addresses

To read a single attribute value using the system service and run-time library routines, use the following routines:

- DNS$_ENUMERATE_OBJECTS function code to enumerate objects
- DNS$REMOVE_FIRST_SET_VALUE run-time library routine to remove the first set value
- DNS$_READ_ATTRIBUTE function code to read the first set value

You can also use the system service and run-time library routines together to add an opaque simple name to a full name by performing the following steps:

1. Obtain a string full name from a user.
2. Use the system service DNS$_PARSE_FULLNAME_STRING function code to convert the string name to opaque format.
3. Use the DNS$_APPEND_SIMPLE_TO_RIGHT run-time library routine to add an opaque simple name to the end of the full name.

11.2 Using the SYS$DNS System Service Call

The following sections describe how to create and modify an object, and then how to read attributes and enumerate names and attributes in the namespace.

Each section contains a code example. These code examples are all contained in the sample program that resides on your distribution medium under the file name SYS$EXAMPLES:SYS$DNS_SAMPLE.C.

11.2.1 Creating Objects

Applications that use DECdns can create an object in the namespace for each resource used by the application. You can create objects using either the SYS$DNS or the SYS$DNSW system service.

A DECdns object consists of a name and its associated attributes. When you create the object, you must assign a class and a class version. You can modify the object to hold additional attributes, such as class-specific attributes, on an as-needed basis.

Note that applications can use objects that are created by other applications.

To create an object in the namespace with SYS$DNS:

1. Prompt the user for a name.

   The name that an application assigns to an object should come from a user, a configuration file, a system logical name, or some other source. The application never assigns an object's name because the namespace structure is uncertain. The name the application receives from the user is in string format.

2. Use the SYS$DNS parse function to convert the full name string into an opaque format. Specify the DNS$_NEXTCHAR_PTR item code to obtain the length of the opaque name.
3. Optionally, reserve an event flag so you can check for completion of the service.
4. Build an item list that contains the following elements:
   - The opaque name for the object (resulting from the translation in step 2)
   - The class name given by the application, which should contain the facility code
   - The class version assigned by the application
   - An optional timeout value that specifies when the call expires
5. Optionally, provide the address of the DECdns status block to receive status information from the name service.
6. Optionally, provide the address of the asynchronous system trap (AST) service routine. AST routines allow a program to continue execution while waiting for parts of the program to complete.
7. Optionally, supply a parameter to pass to the AST routine.
8. Call the create object function and provide all the parameters supplied in steps 1 through 7.

If a clerk call is not complete when timeout occurs, then the call completes with an error. The error is returned in the DECdns status block.

An application should check for errors that are returned; it is not enough to check the return of the SYS$DNS call itself. You need to check the DECdns status block to be sure no errors are returned by the DECdns server.

The following routine, written in C, shows how to create an object in the namespace with the synchronous service SYS$DNSW. The routine demonstrates how to construct an item list.

```
#include <dnsdef.h>
#include <dnsmsg.h>
/*
 * Parameters:
 *      class_name = address of the opaque simple name of the class
 *                   to assign to the object
 *      class_len  = length (in bytes) of the class opaque simple name
 *      object_name= address of opaque full name of the object
 *                   to create in the namespace.
 *      object_len = length (in bytes) of the opaque full name of the
 *                   object to create
 */

create_object(class_name, class_len, object_name, object_len)
unsigned char *class_name;  /*Format is a DECdns opaque simple name*\
unsigned short class_len;
unsigned char *object_name; /*Format is a DECdns opaque simple name*\
unsigned short object_len;
{
    struct $dnsitmdef createitem[4]; /* Item list used by system service */
    struct $dnscversdef version;     /* Version assigned to the object */
    struct $dnsb iosb;          /* Used to determine DECdns server status */
    int status;                 /* Status return from system service */
    /*
     * Construct the item list that creates the object:
     */
    createitem[0].dns$w_itm_size = class_len;   1
    createitem[0].dns$w_itm_code = dns$_class;
    createitem[0].dns$a_itm_address = class_name;

    createitem[1].dns$w_itm_size = object_len;   2
    createitem[1].dns$w_itm_code = dns$_objectname;
    createitem[1].dns$a_itm_address = object_name;

    version.dns$b_c_major = 1;   3
    version.dns$b_c_minor = 0;

    createitem[2].dns$w_itm_size = sizeof(struct $dnscversdef);  4
    createitem[2].dns$w_itm_code = dns$_version;
    createitem[2].dns$a_itm_address = &version;

    *((int *)&createitem[3]) = 0;   5

    status = sys$dnsw(0, dns$_create_object, &createitem, &iosb, 0, 0); 6

    if(status == SS$_NORMAL)
    {
        status = iosb.dns$l_dnsb_status; 7
    }

    return(status);
}
```

1 The first entry in the item list is the address of the opaque simple name that represents the class of the object.

2 The second entry is the address of the opaque full name for the object.

3 The next step is to build a version structure, which will indicate the version of the object. In this case, the object is version 1.0.

4 The third entry is the address of the version structure that was just built.

5 A value of 0 terminates the item list.

6 The next step is to call the system service to create the object.

7 Check to see that both the system service and DECdns were able to perform the operation without error.

11.2.2 Modifying Objects and Their Attributes

After you create objects that identify resources, you can add or modify attributes that describe properties of the object. There is no limit imposed on the number of attributes an object can have.

You modify an object whenever you need to add an attribute or attribute value, change an attribute value, or delete an attribute or attribute value. When you modify an attribute, DECdns updates the timestamp contained in the DNS$UTS attribute for that attribute.

To modify an attribute or attribute value, use the DNS$_MODIFY_ATTRIBUTE function code. Specify the attribute name in the input item code along with the following required input item codes:

- DNS$_ATTRIBUTETYPE to specify a set-valued (DNS$K_SET) or single-valued (DNS$K_SINGLE) attribute
- DNS$_MODOPERATION to specify that the value is being added (DNS$K_PRESENT) or deleted (DNS$K_ABSENT)

Use the DNS$_MODVALUE item code to specify the value of the attribute. Note that the DNS$_MODVALUE item code must be specified to add a single-valued attribute. You can specify a null value for a set-valued attribute. DECdns modifies attribute values in the following way:

- If the attribute exists and you specify an attribute value, the attribute value is removed from a set-valued attribute. All other values are unaffected. For a single-valued attribute, DECdns removes the attribute and its value from the name.
- If you do not specify an attribute value, DECdns removes the attribute and all values of the attribute for both set-valued and single-valued attributes.

To delete an attribute, use the DNS$_MODOPERATION item code.

The following is an example of how to use the DNS$_MODIFY_ATTRIBUTE function code to add a new member to a group object. To do this, you add the new member to the DNS$Members attribute of the group object. Use the following function codes:

- Specify the group object (DNS$_ENTRY) and type (DNS$_LOOKINGFOR). The type should be specified as object (DNS$K_OBJECT).
- Use DNS$_MODOPERATION to add a member to the DNS$Members attribute (DNS$_ATTRIBUTENAME), which is a set-valued attribute (DNS$_ATTRIBUTETYPE).
- Specify the new member object name in DNS$_MODVALUE.
- Use another DNS$_MODIFY_ATTRIBUTE call to assign access rights for the new member to the DNS$ACS attribute of the member object.

Perform the following steps to modify an object with SYS$DNSW:

1. Build an item list that contains the following elements:
   - Opaque name of the object you are modifying
   - Type of object
   - Operation to perform
   - Type of attribute you are modifying
   - Attribute name
   - Value being added to the attribute
2. Supply any of the optional parameters described in Section 11.2.1.
3. Call the modify attribute function, supplying the parameters established in steps 1 and 2.

The following example, written in C, shows how to add a set-valued attribute and a value to an object:

```
#include <dnsdef.h>
#include <dnsmsg.h>
/*
 * Parameters:
 *      obj_name = address of opaque full name of object
 *      obj_len  = length of opaque full name of object
 *      att_name = address of opaque simple name of attribute to create
 *      att_len  = length of opaque simple name of attribute
 *      att_value= value to associate with the attribute
 *      val_len  = length of added value (in bytes)
 */

add_attribute(obj_name, obj_len, att_name, att_len, att_value, val_len)
unsigned char *obj_name;
unsigned short obj_len;
unsigned char *att_name;
unsigned short att_len;
unsigned char *att_value;
unsigned short val_len;

main() {
        struct $dnsitmdef moditem[7];          /* Item list for $DNSW */
        unsigned char objtype = dns$k_object;  /* Using objects */
        unsigned char opertype = dns$k_present; /* Adding an object */
        unsigned char attype = dns$k_set;     /* Attribute will be type set */
        struct $dnsb iosb;                /* Used to determine DECdns status */
        int status;                       /* Status of system service */

    /*
     * Construct the item list to add an attribute to an object.
     */
    moditem[0].dns$w_itm_size = obj_len;
    moditem[0].dns$w_itm_code = dns$_entry;
    moditem[0].dns$a_itm_address = obj_name;   1

    moditem[1].dns$w_itm_size = sizeof(char);
    moditem[1].dns$w_itm_code = dns$_lookingfor;
    moditem[1].dns$a_itm_address = &objtype;   2

    moditem[2].dns$w_itm_size = sizeof(char);
    moditem[2].dns$w_itm_code = dns$_modoperation;
    moditem[2].dns$a_itm_address = &opertype;   3

    moditem[3].dns$w_itm_size = sizeof(char);
    moditem[3].dns$w_itm_code = dns$_attributetype;
    moditem[3].dns$a_itm_address = &attype;   4

    moditem[4].dns$w_itm_size = att_len;
    moditem[4].dns$w_itm_code = dns$_attributename;
    moditem[4].dns$a_itm_address = att_name;   5

    moditem[5].dns$w_itm_size = val_len;
    moditem[5].dns$w_itm_code = dns$_modvalue;
    moditem[5].dns$a_itm_address = att_value;   6

    *((int *)&moditem[6]) = 0;   7

    /*
     * Call $DNSW to add the attribute to the object.
     */
    status = sys$dnsw(0, dns$_modify_attribute, &moditem, &iosb, 0, 0);8
    if(status == SS$_NORMAL)
    {
    status = iosb.dns$l_dnsb_status;  9
    }

    return(status);
}
```

1 The first entry in the item list is the address of the opaque full name of the object.

2 The second entry shows that this is an object, not a soft link or child directory pointer.

3 The third entry is the operation to perform. The program adds an attribute with its value to the object.

4 The fourth entry is the attribute type. The attribute has a set of values rather than a single value.

5 The fifth entry is the opaque simple name of the attribute being added.

6 The sixth entry is the value associated with the attribute.

7 A value of 0 terminates the item list.

8 A call is made to the SYS$DNSW system service to perform the operation.

9 A check is made to see that both the system service and DECdns performed the operation without error.

11.2.3 Requesting Information from DECdns

Once an application adds its objects to the namespace and modifies the names to contain all necessary attributes, the application is ready to use the namespace. An application can request that the DECdns Clerk either read attribute information stored with an object or list all the application's objects that are stored in a particular directory. An application might also need to resolve all soft links in a name in order to identify a target.

To request information from DECdns, use the read or enumerate function codes, as follows:

- The DNS$_READ_ATTRIBUTE function reads and returns a set whose members are the values of the specified attribute.
- The DNS$_ENUMERATE functions return a list of names for attributes, child directories, objects, and soft links.

11.2.3.1 Using the Distributed File Service (DFS)

The VAX Distributed File Service (DFS) uses DECdns for resource naming. This section gives an example of the DNS$_READ_ATTRIBUTE call as used by DFS. The DFS application uses DECdns to give the operating system's users the ability to use remote operating system disks as if the disks were attached to their local VAX system. The DFS application creates DECdns names for the operating system's directory structures (a directory and all of its subdirectories). Each DFS object in the namespace references a particular file access point. DFS creates each object with a class attribute of DFS$ACCESSPOINT and modifies the address attribute (DNS$Address) of each object to hold the DECnet node address where the directory structures reside. As a final step in registering its resources, DFS creates a database that maps DECdns names to the appropriate operating system directory structures.

Whenever the DFS application receives the following mount request, DFS sends a request for information to the DECdns Clerk:

MOUNT ACCESS_POINT dns-name vms-logical-name

To read the address attribute of the access point object, the DFS application performs the following steps:

1. Translates the DECdns name that is supplied through the user to opaque format using the SYS$DNS parse function
2. Reads the class attribute of the object with the $DNS read attribute function, indicating that there is a second call to read other attributes of the object
3. Makes a second call to the SYS$DNS read attribute function to read the address attribute of the object
4. Sends the DECdns name to the DFS server, which looks up the disk on which the access point is located
5. Verifies that the DECdns name is valid on the DFS server

The DFS client and DFS server now can communicate to complete the mount function.

11.2.3.2 Reading Attributes from DNS

When requesting information from DNS, an application always takes an object name from the user, translates the name into opaque format, and passes it in an item list to the DECdns Clerk.

Each read request returns a set of attribute values. The DNS$_READ_ATTRIBUTE service uses a context item code called DNS$_CONTEXTVARTIME to maintain context when reading the attribute values. The context item code saves the last member that is read from the set. When the next read call is issued, the item code sets the context to the next member in the set, reads it, and returns it. The context item code treats single-valued attributes as though they were a set of one.

If an enumeration call returns DNS$_MOREDATA, not all matching names or attributes have been enumerated. If you receive this message, you should make further calls, setting DNS$_CONTEXTVARTIME to the last value returned until the procedure returns SS$_NORMAL.

The following program, written in C, shows how an application reads an object attribute. The SYS$DNSW service uses an item list to return a set of objects. Then the application calls a run-time library routine to read each value in the set.

```
#include <dnsdef.h>
#include <dnsmsg.h>
/*
 * Parameters:
 *      opaque_objname = address of opaque full name for the object
 *                       containing the attribute to be read
 *      obj_len        = length of opaque full name of the object
 *      opaque_attname = address of the opaque simple name of the
 *                       attribute to be read
 *      attname_len    = length of opaque simple name of attribute
 */
```

```
read_attribute(opaque_objname, obj_len, opaque_attname, attname_len)
unsigned char *opaque_objname;
unsigned short obj_len;
unsigned char *opaque_attname;
unsigned short attname_len;
{
    struct $dnsb iosb;             /* Used to determine DECdns status */
    char objtype = dns$k_object;   /* Using objects */

    struct $dnsitmdef readitem[6]; /* Item list for system service */
    struct dsc$descriptor set_dsc, value_dsc, newset_dsc, cts_dsc;

    unsigned char attvalbuf[dns$k_maxattribute]; /* To hold the attribute */
                             /* values returned from extraction routine. */
    unsigned char attsetbuf[dns$k_maxattribute]; /* To hold the set of    */
                         /* attribute values after the return from $DNSW. */
    unsigned char ctsbuf[dns$k_cts_length];     /* Needed for context of multiple reads */

    int read_status;         /* Status of read attribute routine */
    int set_status;          /* Status of remove value routine */
    int xx;                  /* General variable used by print routine */

    unsigned short setlen;   /* Contains current length of set structure */
    unsigned short val_len;  /* Contains length of value extracted from set */
    unsigned short cts_len;  /* Contains length of CTS extracted from set */

    /* Construct an item list to read values of the attribute. */ 1
    readitem[0].dns$w_itm_code = dns$_entry;
    readitem[0].dns$w_itm_size = obj_len;
    readitem[0].dns$a_itm_address = opaque_objname;

    readitem[1].dns$w_itm_code = dns$_lookingfor;
    readitem[1].dns$w_itm_size = sizeof(char);
    readitem[1].dns$a_itm_address = &objtype;

    readitem[2].dns$w_itm_code = dns$_attributename;
    readitem[2].dns$a_itm_address = opaque_attname;
    readitem[2].dns$w_itm_size = attname_len;

    readitem[3].dns$w_itm_code = dns$_outvalset;
    readitem[3].dns$a_itm_ret_length = &setlen;
    readitem[3].dns$w_itm_size = dns$k_maxattribute;
    readitem[3].dns$a_itm_address = attsetbuf;

    *((int *)&readitem[4]) = 0;

    do   2
    {
        read_status = sys$dnsw(0, dns$_read_attribute, &readitem, &iosb, 0, 0);

        if(read_status == SS$_NORMAL)
        {
            read_status = iosb.dns$l_dnsb_status;
        }

        if((read_status == SS$_NORMAL) || (read_status == DNS$_MOREDATA))
        {
            do
            {
                set_dsc.dsc$w_length = setlen;
                set_dsc.dsc$a_pointer = attsetbuf; /* Address of set */

                value_dsc.dsc$w_length = dns$k_simplenamemax;
                value_dsc.dsc$a_pointer = attvalbuf;  /* Buffer to hold  */
                                                      /* attribute value */

                cts_dsc.dsc$w_length = dns$k_cts_length;
                cts_dsc.dsc$a_pointer = ctsbuf; /* Buffer to hold value's CTS*/
                newset_dsc.dsc$w_length = dns$k_maxattribute;
                newset_dsc.dsc$a_pointer = attsetbuf; /* Same buffer for */
                                                      /* each call      */

                set_status = dns$remove_first_set_value(&set_dsc, &value_dsc,
                            3                 &val_len, &cts_dsc,
                                                     &cts_len, &newset_dsc,
                                                     &setlen);

                if(set_status == SS$_NORMAL)
                {   4
                    readitem[4].dns$w_itm_code = dns$_contextvartime;
                    readitem[4].dns$w_itm_size = cts_len;
                    readitem[4].dns$a_itm_address = ctsbuf;

                    *((int *)&readitem[5]) = 0;

                    printf("\tValue: ");   5
                    for(xx = 0; xx < val_len; xx++)
                        printf("%x ", attvalbuf[xx]);
                    printf("\n");
                }
                else if (set_status != 0)
                {
                    printf("Error %d returned when removing value from set\n",
                           set_status);
                    exit(set_status);
                }
            } while(set_status == SS$_NORMAL);
        }
        else
        {
            printf("Error reading attribute = %d\n", read_status);
            exit(read_status);
        }
    } while(read_status == DNS$_MOREDATA);
}
```

1 The item list contains five entries:

- Opaque full name of the object with the attribute the program wants to read
- Type of object to access
- Opaque simple name of the attribute to read
- Address of the buffer containing the set of values returned by the read operation
- A value of 0 to terminate the item list

2 The loop repeatedly calls the SYS$DNSW service to read the values of the attribute because the first call might not return all the values. The loop executes until $DNSW returns something other than DNS$_MOREDATA.

3 The DNS$REMOVE_FIRST_SET_VALUE routine extracts a value from the set.

4 This attribute name may be the context the routine uses to read additional attributes. The attribute's creation timestamp (CTS), not its value, provides the context.

5 Finally, display the value in hexadecimal format. (You could also take the attribute name and convert it to a printable format before displaying the result.)

See the discussion about setting confidence in the *Guide to Programming with DECdns* for information about obtaining up-to-date data on read requests.

11.2.3.3 Enumerating DECdns Names and Attributes

The enumerate functions return DECdns names for objects, child directories, soft links, groups, or attributes in a specific directory. Use either the asterisk (*) or question mark (?) wildcard to screen enumerated items. DECdns matches any single character against the specified wildcard.

Enumeration calls return a set of simple names or attributes. If an enumeration call returns DNS$_MOREDATA, not all matching names or attributes have been enumerated. If you receive this message, use the context-setting conventions that are described for the DNS$_READ_ATTRIBUTE call. You should make further calls, setting DNS$_CONTEXTVARNAME to the last value returned until the procedure returns SS$_NORMAL. For more information, see the SYS$DNS system service in the *OpenVMS System Services Reference Manual: A–GETMSG*.

The following program, written in C, shows how an application can read the objects in a directory with the SYS$DNS system service. The values that DECdns returns from read and enumerate functions are in different structures. For example, an enumeration of objects returns different structures than an enumeration of child directories. To clarify how to use this data, the sample program demonstrates how to parse any set that the enumerate objects function returns with a run-time library routine in order to remove the first value from the set. The example also demonstrates how the program takes each value from the set.

```
#include <dnsdef.h>
#include <dnsmsg.h>
/*
 * Parameters:
 *      fname_p   : opaque full name of the directory to enumerate
 *      fname_len : length of full name of the directory
 */

struct $dnsitmdef enumitem[4];            /* Item list for enumeration */
unsigned char setbuf[100];                /* Values from enumeration */
struct $dnsb enum_iosb;       /* DECdns status information */
int synch_event;              /* Used for synchronous AST threads */
unsigned short setlen;        /* Length of output in setbuf */

enumerate_objects(fname_p, fname_len)
unsigned char *fname_p;
unsigned short fname_len;
{
    int enumerate_objects_ast();

    int status;             /* General routine status */
    int enum_status;        /* Status of enumeration routine */

    /* Set up item list */

    enumitem[0].dns$w_itm_code = dns$_directory; /* Opaque directory name */
    enumitem[0].dns$w_itm_size = fname_len;
    enumitem[0].dns$a_itm_address = fname_p;

    enumitem[1].dns$w_itm_code = dns$_outobjects;  /* output buffer */
    enumitem[1].dns$a_itm_ret_length = &setlen;
    enumitem[1].dns$w_itm_size = 100;
    enumitem[1].dns$a_itm_address = setbuf;

    *((int *)&enumitem[2]) = 0; /* Zero terminate item list */

    status = lib$get_ef(&synch_event);  1
    if(status != SS$_NORMAL)
    {
        printf("Could not get event flag to synch AST threads\n");
        exit(status);
    }

    enum_status = sys$dns(0, dns$_enumerate_objects, &enumitem,
             2     &enum_iosb, enumerate_objects_ast, setbuf);

    if(enum_status != SS$_NORMAL)   3
    {
        printf("Error enumerating objects = %d\n", enum_status);
        exit(enum_status);
    }
    status = sys$synch(synch_event, &enum_iosb);  4

    if(status != SS$_NORMAL)
    {
        printf("Synchronization with AST threads failed\n");
        exit(status);
    }
}

/* AST routine parameter:                       */
/*      outbuf : address of buffer that contains enumerated names. */
                                                     5
unsigned char objnamebuf[dns$k_simplenamemax]; /* Opaque object name */

enumerate_objects_ast(outbuf)
unsigned char *outbuf;
{
    struct $dnsitmdef cvtitem[3];             /* Item list for class name */
    struct $dnsb iosb;        /* Used for name service status information */
    struct dsc$descriptor set_dsc, value_dsc, newset_dsc;

    unsigned char simplebuf[dns$k_simplestrmax];   /* Object name string */

    int enum_status;   /* The status of the enumeration itself */
    int status;        /* Used for checking immediate status returns */
    int set_status;    /* Status of remove value routine */

    unsigned short val_len;    /* Length of set value */
    unsigned short sname_len;  /* Length of object name */

    enum_status = enum_iosb.dns$l_dnsb_status;  /* Check status */
    if((enum_status != SS$_NORMAL) && (enum_status != DNS$_MOREDATA))
    {
        printf("Error enumerating objects = %d\n", enum_status);
        sys$setef(synch_event);
        exit(enum_status);
    }

    do
    {
        /*
         * Extract object names from output buffer one
         * value at a time.  Set up descriptors for the extraction.
         */
        set_dsc.dsc$w_length = setlen;    /* Contains address of */
        set_dsc.dsc$a_pointer = setbuf;   /* the set whose values */
                                          /* are to be extracted */

        value_dsc.dsc$w_length = dns$k_simplenamemax;
        value_dsc.dsc$a_pointer = objnamebuf; /* To contain the */
                                              /* name of an object */
                                              /* after the extraction */

        newset_dsc.dsc$w_length = 100;      /* To contain a new */
        newset_dsc.dsc$a_pointer = setbuf;  /* set structure after */
                                            /* the extraction. */
```

```
            /* Call yRTL routine to extract the value from the set */
            set_status = dns$remove_first_set_value(&set_dsc, &value_dsc, &val_len,
                                                    0, 0, &newset_dsc, &setlen);

            if(set_status == SS$_NORMAL)
            {                                      6
                cvtitem[0].dns$w_itm_code = dns$_fromsimplename;
                cvtitem[0].dns$w_itm_size = val_len;
                cvtitem[0].dns$a_itm_address = objnamebuf;

                cvtitem[1].dns$w_itm_code = dns$_tostringname;
                cvtitem[1].dns$w_itm_size = dns$k_simplestrmax;
                cvtitem[1].dns$a_itm_address = simplebuf;
                cvtitem[1].dns$a_itm_ret_length = &sname_len;

                *((int *)&cvtitem[2]) = 0;

                status = sys$dnsw(0, dns$_simple_opaque_to_string, &cvtitem,
                                  &iosb, 0, 0);

                if(status == SS$_NORMAL)
                    status = iosb.dns$l_dnsb_status;  /* Check for errors */

                if(status != SS$_NORMAL) /* If error, terminate processing */
                {
                    printf("Converting object name to string returned %d\n",
                           status);
                    exit(status);
                }
                else
                {
                    printf("%.*s\n", sname_len,simplebuf);
                }

                enumitem[2].dns$w_itm_code = dns$_contextvarname;   7
                enumitem[2].dns$w_itm_size = val_len;
                enumitem[2].dns$a_itm_address = objnamebuf;

                *((int *)&enumitem[3]) = 0;
            }
            else if (set_status != 0)
            {
                printf("Error %d returned when removing value from set\n",
                       set_status);
                exit(set_status);
            }
        } while(set_status == SS$_NORMAL);

        if(enum_status == DNS$_MOREDATA)
        {                                          8
            enum_status = sys$dns(0, dns$_enumerate_objects, &enumitem,
                                  &enum_iosb, enumerate_objects_ast, setbuf);

            if(enum_status != SS$_NORMAL)  /* Check status of $DNS */
            {
                printf("Error enumerating objects = %d\n", enum_status);
                sys$setef(synch_event);
            }
        }
        else
        {                                          9
            sys$setef(synch_event);
        }
}
```

1 Get an event flag to synchronize the execution of AST threads.

2 Use the system service to enumerate the object names.

3 Check the status of the system service itself before waiting for threads.

4 Use the SYS$SYNCH call to make sure the DECdns Clerk has completed and that all threads have finished executing.

5 After enumerating objects, SYS$DNS calls an AST routine. The routine shows how DNS$REMOVE_FIRST_SET_VALUE extracts object names from the set returned by the DNS$_ENUMERATE_OBJECTS function.

6 Use an item list to convert the opaque simple name to a string name so you can display it to the user. The item list contains the following entries:

- Address of the opaque simple name to be converted
- Address of the buffer that will hold the string name
- A value of 0 to terminate the item list

7 This object name may provide the context for continuing the enumeration. Append the context variable to the item list so the enumeration can continue from this name if there is more data.

8 Use the system service to enumerate the object names as long as there is more data.

9 Set the event flag to indicate that all AST threads have completed and that the program can terminate.

11.3 Using the DCL Command DEFINE with DECdns Logical Names

When the DECdns Clerk is started on the operating system, the VAX system creates a unique logical name table for DECdns to use in translating full names. This logical name table, called DNS$SYSTEM, prevents unintended interaction with other system logical names.

To define systemwide logical names for DECdns objects, you must have the appropriate privileges to use the DCL command DEFINE. Use the DEFINE command to create the logical RESEARCH.PROJECT_DISK, for example, by entering the following DCL command:

```
$ DEFINE/TABLE=DNS$SYSTEM RESEARCH "ENG.RESEARCH"
```

When parsing a name, the SYS$DNS service specifies the logical name DNS$LOGICAL as the table it uses to translate a simple name into a full name. This name translates to DNS$SYSTEM (by default) to access the systemwide DECdns logical name table.

To define process or job logical names for SYS$DNS, you must create a process or job table and redefine DNS$LOGICAL as a search list, as in the following example (note that elevated privileges are required to create a job table):

```
$ CREATE /NAME_TABLE DNS_PROCESS_TABLE
$ DEFINE /TABLE=LNM$PROCESS_DIRECTORY DNS$LOGICAL -
_$DNS_PROCESS_TABLE,DNS$SYSTEM
```

Once you have created the process or job table and redefined DNS$LOGICAL, you can create job-specific logical names for DECdns by using the DCL command DEFINE, as follows:

```
$ DEFINE /TABLE=DNS_PROCESS_TABLE RESEARCH "ENG.RESEARCH.MYGROUP"
```

12
Using the Distributed Transaction Manager

This chapter describes how to use the distributed transaction manager. It shows you how to use DECdtm services to bind together operations on several databases or files into a single transaction. To use DECdtm services, the resource managers taking part in the transaction must support DECdtm. DEC Rdb for OpenVMS Alpha, DEC Rdb for OpenVMS VAX, DEC DBMS for OpenVMS Alpha, DEC DBMS for OpenVMS VAX, and OpenVMS RMS Journaling support DECdtm.

This chapter is divided into the following sections:

Section 12.1 gives an introduction to DECdtm services.

Section 12.2 discusses how to call DECdtm services.

Section 12.3 gives an example that shows how to use DECdtm services.

12.1 Introduction to DECdtm Services

A transaction performs operations on resources. Examples of resources are databases and files. A transaction often needs to use more than one resource on one or more nodes. This type of transaction is called a **distributed transaction**.

Maintaining the integrity and consistency of the resources used by a distributed transaction can be complex. To help with this, DECdtm manages distributed transactions and reduces the amount of coding required in your applications.

DECdtm uses an optimized version of the standard two-phase commit protocol. This ensures that transactions are **atomic**. If a transaction is atomic, either all the transaction operations take effect (the transaction is **committed**), or none of the operations take effect (the transaction is **aborted**).

The two-phase commit protocol makes sure that *all* the operations can take effect before the transaction is committed. If any operation cannot take effect, for example if a network link is lost, then the transaction is aborted, and none of the operations take effect.

12.1.1 Sample Atomic Transaction

Edward Jessup, an employee of a computer company in Italy, is transferring to a subsidiary of the company in Japan. An application must remove his personal information from an Italian DBMS database and add them to a Japanese Rdb database. Both of these operations must happen, otherwise Edward may either end up “in limbo” (the application might remove him from the Italian database but then lose a network link while trying to add him to the Japanese database), or find that he is in both databases at the same time. Either way, the two databases would be out of step.

If the application used DECdtm to execute both operations as an atomic transaction, then this error could never happen; DECdtm would automatically detect the network link failure and abort the transaction. Neither of the databases would be updated, and the application could then try again.

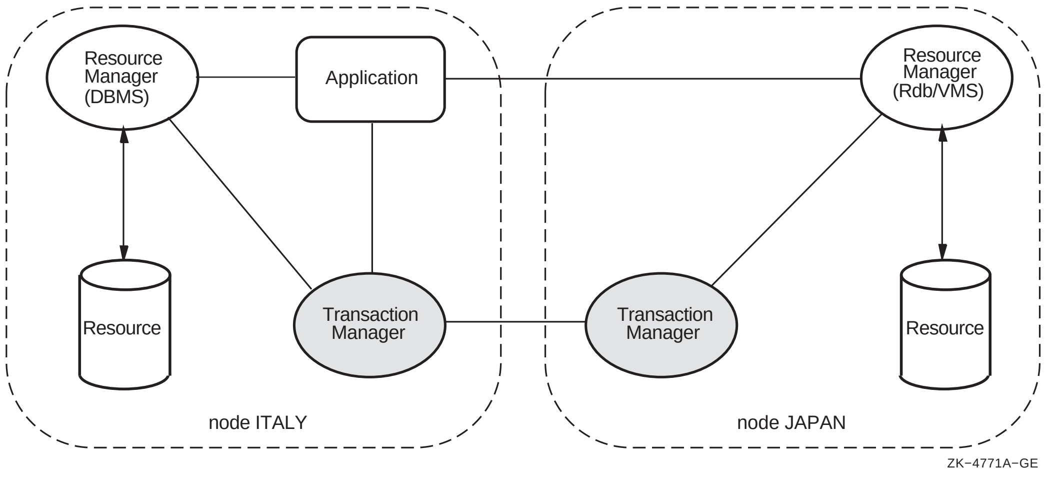
12.1.2 Transaction Participants

A DECdtm transaction involves the following participants:

- **Application**: Defines the operations that the transaction will perform.
- **Resource manager**: Performs the operations on the resources. A resource manager must support DECdtm. Examples of those that do are Rdb, DBMS, and OpenVMS RMS Journaling.
- **Transaction manager**: Coordinates the actions of the resource managers on its node. Transaction managers are provided by DECdtm.

Figure 12–1 shows the participants in the distributed transaction discussed in Section 12.1.1. The application is on node ITALY.

Figure 12–1 Participants in a Distributed Transaction



12.1.3 DECdtm System Services

The DECdtm system services are:

- SYS$START_TRANSW: Starts a new transaction and returns the transaction identifier.
- SYS$END_TRANSW: Ends a transaction by attempting to commit it. Returns the outcome of the transaction (either commit or abort).
- SYS$ABORT_TRANSW: Aborts a transaction.

These are all synchronous system service calls. There are also asynchronous versions (SYS$START_TRANS, SYS$END_TRANS, and SYS$ABORT_TRANS). For a full description of all the DECdtm system services, see the *OpenVMS System Services Reference Manual*.

12.1.4 Default Transactions

Some resource managers (such as OpenVMS RMS Journaling) support the concept of **default transactions**. This means that the application does not need to specify the transaction identifier when executing transaction operations. The resource manager checks whether the calling process has a default transaction; if it has, the resource manager assumes that the operation is part of the default transaction.

12.2 Calling DECdtm System Services

An application using the DECdtm system services follows these steps:

1. Calls SYS$START_TRANSW. This starts a new transaction and returns the transaction identifier.
2. Instructs the resource managers to perform the required operations on their resources.
3. Ends the transaction in one of two ways:
   - **Commit:** To attempt to perform, or commit, the transaction, the application calls SYS$END_TRANSW. This checks whether all the participants can commit their operations. If any participant cannot commit an operation, the transaction is aborted.

     When SYS$END_TRANSW returns, the application finds out the outcome of the transaction by reading the completion status in the I/O status block.
   - **Abort:** To abort the transaction, the application calls SYS$ABORT_TRANSW. Typically, an application aborts a transaction if a resource manager returns an error or if the user enters invalid information during the transaction.

12.3 Using DECdtm Services: An Example

The following is a sample Fortran application that uses DECdtm system services. It can be found in SYS$EXAMPLES:DECDTM$EXAMPLE1.

The application opens two files, sets a counter, then enters a loop to perform the following steps:

- Increments the counter by 1
- Calls SYS$START_TRANSW to start a new transaction
- Writes the counter value to the two files
- Either calls SYS$END_TRANSW to attempt to commit the transaction, or calls SYS$ABORT_TRANSW to abort the transaction.

The application repeats these steps until either an error occurs or the user requests an interrupt. Because DECdtm services are used, the two files will always be in step with each other. If DECdtm services were not used, one file could have been updated while the other was not. This would result in the files' being out of step.

This example contains numbered callouts, which are explained after the program listing.

```
  C
  C This program assumes that the files DECDTM$EXAMPLE1.FILE_1 and
  C DECDTM$EXAMPLE1.FILE_2 are created and marked for recovery unit
  C journaling using the command file SYS$EXAMPLES:DECDTM$EXAMPLE1.COM
  C
  C To run this example, enter the following:
  C   $ FORTRAN SYS$EXAMPLES:DECDTM$EXAMPLE1
  C   $ LINK DECDTM$EXAMPLE1
  C   $ @SYS$EXAMPLES:DECDTM$EXAMPLE1
  C   $ RUN DECDTM$EXAMPLE1
  C
  C SYS$EXAMPLES also contains an example C application, DECDTM$EXAMPLE2.C
  C The C application performs the same operations as this Fortran example.
  C
          IMPLICIT    NONE

          INCLUDE     '($SSDEF)'
          INCLUDE     '($FORIOSDEF)'

          CHARACTER*12 STRING
          INTEGER*2   IOSB(4)
          INTEGER*4   STATUS,COUNT,TID(4)
          INTEGER*4   SYS$START_TRANSW,SYS$END_TRANSW,SYS$ABORT_TRANSW
          EXTERNAL    SYS$START_TRANSW,SYS$END_TRANSW,SYS$ABORT_TRANSW
          EXTERNAL    JOURNAL_OPEN
  C
  C Open the two files
  C
1         OPEN (UNIT = 10, FILE = 'DECDTM$EXAMPLE1.FILE_1', STATUS = 'OLD',
          1     ACCESS = 'DIRECT', RECL = 3, USEROPEN = JOURNAL_OPEN)
          OPEN (UNIT = 11, FILE = 'DECDTM$EXAMPLE1.FILE_2', STATUS = 'OLD',
          1     ACCESS = 'DIRECT', RECL = 3, USEROPEN = JOURNAL_OPEN)

          COUNT = 0

          TYPE *, 'Running DECdtm example program'
          TYPE *, 'Press CTRL-Y to interrupt'
  C
  C Loop forever, updating both files under transaction control
  C
          DO WHILE (.TRUE.)
  C
  C Update the count and convert it to ASCII
  C
2           COUNT = COUNT + 1
            ENCODE (12,8000,STRING) COUNT
  8000      FORMAT (I12)
  C
  C Start the transaction
  C
3           STATUS = SYS$START_TRANSW (%VAL(1),,IOSB,,,TID)
            IF (STATUS .NE. SS$_NORMAL .OR. IOSB(1) .NE. SS$_NORMAL) GO TO 9040
  C
  C Update the record in each file
  C
4           WRITE (UNIT = 10, REC = 1, ERR = 9000, IOSTAT = STATUS) STRING
            WRITE (UNIT = 11, REC = 1, ERR = 9010, IOSTAT = STATUS) STRING
  C
  C Attempt to commit the transaction
  C
5           STATUS = SYS$END_TRANSW (%VAL(1),,IOSB,,,TID)
            IF (STATUS .NE. SS$_NORMAL .OR. IOSB(1) .NE. SS$_NORMAL) GO TO 9050
```

```
        END DO
C
C Errors that should cause the transaction to abort
C
6
9000    TYPE *, 'Failed to update DECDTM$EXAMPLE1.FILE_1'
        GO TO 9020

9010    TYPE *, 'Failed to update DECDTM$EXAMPLE1.FILE_2'
9020    STATUS = SYS$ABORT_TRANSW (%VAL(1),,IOSB,,,TID)
        IF (STATUS .NE. SS$_NORMAL .OR. IOSB(1) .NE. SS$_NORMAL) GO TO 9060
        STOP
C
C Errors from DECdtm system services
C
9040    TYPE *, 'Unable to start a transaction'
        GO TO 9070
9050    TYPE *, 'Failed to commit the transaction'
        GO TO 9070
9060    TYPE *, 'Failed to abort the transaction'
9070    TYPE *, 'Status = ', STATUS, ' IOSB = ', IOSB(1)
        END
C
C Switch off TRUNCATE access and PUT with truncate on OPEN for RU Journaling
C
        INTEGER FUNCTION JOURNAL_OPEN (FAB, RAB, LUN)

        INCLUDE '($FABDEF)'
        INCLUDE '($RABDEF)'
        INCLUDE '($SYSSRVNAM)'

        RECORD  /FABDEF/ FAB, /RABDEF/ RAB

        FAB.FAB$B_FAC = FAB.FAB$B_FAC .AND. .NOT. FAB$M_TRN
        RAB.RAB$L_ROP = RAB.RAB$L_ROP .AND. .NOT. RAB$M_TPT

        JOURNAL_OPEN = SYS$OPEN (FAB)
        IF (.NOT. JOURNAL_OPEN) RETURN
        JOURNAL_OPEN = SYS$CONNECT (RAB)

        RETURN
        END
```

1 The application opens DECDTM$EXAMPLE1.FILE1 and DECDTM$EXAMPLE1.FILE2 for writing. It then zeroes the variable COUNT and enters an infinite loop.

2 The application increments the count by one and converts it to an ASCII string.

3 The application calls SYS$START_TRANSW to start a transaction. The application checks the immediate return status and service completion status to see whether they signify an error.

4 The application attempts to write the string to the two files. If it cannot, the application aborts the transaction. Because the files are OpenVMS RMS journaled files, the default transaction is assumed.

5 The application calls SYS$END_TRANSW to attempt to commit the transaction. It checks the immediate return status and service completion status to see whether they signify an error. If they do, the application reports the error and exits. If there are no errors, the transaction is committed and the application continues with the loop.

6 If either of the two files could not be updated, the application calls SYS$ABORT_TRANSW to abort the transaction. It checks the immediate return status and service completion status to see whether they signify an error. If they do, the application reports the error and exits.

13 Condition-Handling Routines and Services

This chapter describes the OpenVMS Condition Handling facility. It contains the following sections:

Section 13.1 gives an overview of run-time errors.

Section 13.2 gives an overview of the OpenVMS Condition Handling facility, presenting condition-handling terminology and functionality.

Section 13.3 describes VAX system and Alpha system exceptions, arithmetic exceptions, and unaligned access traps on Alpha systems.

Section 13.4 describes how run-time library routines handle exceptions.

Section 13.5 describes the condition value field and the testing and modifying of values.

Section 13.6 describes the exception dispatcher.

Section 13.7 describes the argument list that is passed to a condition handler.

Section 13.8 describes signaling.

Section 13.9 describes types of condition handlers.

Section 13.10 describes types of actions performed by condition handlers.

Section 13.11 describes messages and how to use them.

Section 13.12 describes how to write a condition handler.

Section 13.13 describes how to debug a condition handler.

Section 13.14 describes several run-time library routines that can be established as condition handlers.

Section 13.15 describes how to establish, write, and debug an exit handler.

13.1 Overview of Run-Time Errors

Run-time errors are hardware- or software-detected events, usually errors, that alter normal program execution. Examples of run-time errors are as follows:

- System errors—for example, specifying an invalid argument to a system-defined procedure
- Language-specific errors—for example, in 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 operating system either returns a condition code or value identifying the error to your program or signals the condition code. If the operating system signals the condition code, typically an error message is displayed, and program execution continues or terminates, depending on the severity of the error. See Section 13.5 for details about condition values.

When unexpected errors occur, your program should display a message identifying 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.

Both an error message and its associated condition code identify an error by the name of the facility that generated it and an abbreviation of the message text. Therefore, if your program displays an error message, you can identify the condition code that was signaled. For example, if your program displays the following error message, you know that the condition code SS$_NOPRIV was signaled:

```
%SYSTEM-F-NOPRIV, no privilege for attempted operation
```

13.2 Overview of the OpenVMS Condition Handling facility

The operating system provides a set of signaling and condition-handling routines and related system services to handle exception conditions. This set of services is called the OpenVMS Condition Handling facility (CHF). The OpenVMS Condition Handling Facility is a part of the common run-time environment of OpenVMS, which includes run-time library (RTL) routines and other components of the operating system.

The OpenVMS Condition Handling facility provides a single, unified method to enable condition handlers, signal conditions, print error messages, change the error behavior from the system default, and enable or disable detection of certain hardware errors. The RTL and all layered products of the operating sytem use the CHF for condition handling.

See the *OpenVMS Calling Standard* for a detailed description of OpenVMS condition handling.

13.2.1 Condition-Handling Terminology

This section defines terms used to describe condition handling.

exception
An event detected by the hardware or software that changes the normal flow of instruction execution. An exception is a synchronous event caused by the execution of an instruction and often means something generated by hardware. When an exception occurs, the processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some exceptions are relevant primarily to the current process and normally invoke software in the context of the current process. An integer overflow exception detected by the hardware is an example of an event that is reported to the process. Other exceptions, such as page faults, are handled by the operating system and are transparent to the user.

An exception may also be signaled by a routine (software signaling) by calling the RTL routines LIB$SIGNAL or LIB$STOP.

condition

An informational state that exists when an exception occurs. *Condition* is a more general term than *exception*; a condition implies either a hardware exception or a software-raised condition. Often, the term condition is preferred because the term exception implies an error. Section 13.3.1 and Section 13.3.1.1 further define the differences between exceptions and conditions.

condition handling

When a condition is detected during the execution of a routine, a signal can be raised by the routine. The routine is then permitted to respond to the condition. The routine's response is called **handling the condition.**

VAX

On VAX systems, an address of 0 in the first longword of a procedure call frame or in an exception vector indicates that a condition handler does not exist for that call frame or vector. ♦

Alpha

On Alpha systems, the handler valid flag bit in the procedure descriptor is cleared to indicate that a condition handler does not exist. ♦

The condition handlers are themselves routines; they have their own call frames. Because they are routines, condition handlers can have condition handlers of their own. This allows condition handlers to field exceptions that might occur within themselves in a modular fashion.

VAX

On VAX systems, a routine can enable a condition handler by placing the address of the condition handler in the first longword of its stack frame. ♦

Alpha

On Alpha systems, the association of a handler with a procedure is static and must be specified at the time a procedure is compiled (or assembled). Some languages that lack their own exception-handling syntax, however, may support emulation of dynamic specified handlers by means of built-in routines. ♦

If you determine that a program needs to be informed of particular exceptions so it can take corrective action, you can write and specify a condition handler. This condition handler, which receives control when any exception occurs, can test for specific exceptions.

If an exception occurs and you have not specified a condition handler, the default condition handler established by the operating system is given control. If the exception is a fatal error, the default condition handler issues a descriptive message and causes the image that incurred the exception to exit.

To declare or enable a condition handler, use the following system services:

- Set Exception Vector (SYS$SETEXV)
- Set System Service Failure Exception Mode (SYS$SETSFM)
- Unwind from Condition Handler Frame (SYS$UNWIND)
- Declare Change Mode or Compatibility Mode Handler (SYS$DCLCMH)

Parallel mechanisms exist for uniform dispatching of hardware and software exception conditions. Exceptions that are detected and signaled by hardware transfer control to an exception service routine in the executive. Software-detected exception conditions are generated by calling the run-time library routines LIB$SIGNAL or LIB$STOP. Hardware- and software-detected exceptions eventually execute the same exception dispatching code. Therefore, a condition handler may handle an exception condition generated by hardware or by software identically.

The Set Exception Vector (SYS$SETEXV) system service allows you to specify addresses for a primary exception handler, a secondary exception handler, and a last-chance exception handler. You can specify handlers for each access mode. The primary exception vector is reserved for the debugger. In general, you should avoid using these vectored handlers unless absolutely necessary. If you use a vectored handler, it must be prepared for all exceptions occurring in that access mode.

13.2.2 Functions of the Condition Handling facility

The OpenVMS Condition Handling facility and the related run-time library routines and system services perform the following functions:

- Establish and call condition-handler routines

  You can establish condition handlers to receive control in the event of an exception in one of the following ways:

  VAX
  - On VAX systems, by specifying the address of a condition handler in the first longword of a procedure call frame. ♦

    Alpha
    On Alpha systems, the method for establishing a dynamic (that is, nonvectored) condition handler is specified by the language. ♦
  - By establishing exception handlers with the Set Exception Vector (SYS$SETEXV) system service.

  The first of these methods is the preferred way to specify a condition handler for a particular image. The use of dynamic handlers is also the most efficient way in terms of declaration. Vectored handlers should be used for special purposes, such as writing debuggers.

  The VAX MACRO programmer can use the following single-move address instruction to place the address of the condition handler in the longword pointed to by the current frame pointer (FP):

```
MOVAB     HANDLER,(FP)
```

  You can associate a condition handler for the currently executing routine by specifying an address pointing to the handler, either in the routine's stack frame on VAX systems or in one of the exception vectors. (The MACRO-32 compiler for OpenVMS Alpha systems generates the appropriate Alpha code from this VAX instruction to establish a dynamic condition handler.)

  VAX
  On VAX systems, the high-level language programmer can call the common run-time library routine LIB$ESTABLISH (see the *OpenVMS RTL Library (LIB$) Manual*), using the name of the handler as an argument. LIB$ESTABLISH returns as a function value the address of the former handler established for the routine or 0 if no handler existed.

  The new condition handler remains in effect for your routine until you call LIB$REVERT or control returns to the caller of the caller of LIB$ESTABLISH. Once this happens, you must call LIB$ESTABLISH again if the same (or a new) condition handler is to be associated with the caller of LIB$ESTABLISH.

  Some languages provide access to condition handling as part of the language. The ON ERROR GOTO statement in BASIC and the ON statement in PL/I can be used to define condition handlers. If you are using a language that does provide access to condition handling, use its language mechanism rather than LIB$ESTABLISH. Each procedure can declare a condition handler. ♦

When the routine signals an exception, the OpenVMS Condition Handling facility calls the condition handler associated with the routine. See Section 13.8 for more information about exception vectors. Figure 13–5 shows a sample stack scan for a condition handler.

The following DEC Fortran program segment establishes the condition handler ERRLOG. Because the condition handler is used as an actual argument, it must be declared in an EXTERNAL statement.

```
INTEGER*4 OLD_HANDLER
EXTERNAL  ERRLOG
    .
    .
    .
OLD_HANDLER = LIB$ESTABLISH (ERRLOG)
```

As its function value, LIB$ESTABLISH returns the address of the previous handler. If only part of a program unit requires a special condition handler, you can reestablish the original handler by invoking LIB$ESTABLISH and specifying the saved handler address as follows:

```
CALL LIB$ESTABLISH (OLD_HANDLER)
```

The run-time library provides several condition handlers and routines that a condition handler can call. These routines take care of several common exception conditions. Section 13.14 describes these routines.

Alpha

On Alpha systems, LIB$ESTABLISH and LIB$REVERT are not supported, though a high-level language may support them for compatibility. (Table 13–4 lists other run-time library routines supported and not supported on Alpha systems.) ♦

VAX

- Remove an established condition-handler routine

  Using LIB$REVERT, you can remove a condition handler from a routine's stack frame by setting the frame's handler address to 0. If your high-level language provides condition-handling statements, you should use them rather than LIB$REVERT.

- Enable or disable the detection of arithmetic hardware exceptions

  Using run-time library routines, you can enable or disable the signaling of floating point underflow, integer overflow, and decimal overflow, which are detected by the VAX hardware. ♦

- Signal a condition

  When the hardware detects an exception, such as an integer overflow, a signal is raised at that instruction. A routine may also raise a signal by calling LIB$SIGNAL or LIB$STOP. Signals raised by LIB$SIGNAL allow the condition handler either to terminate or to resume the normal flow of the routine. Signals raised by LIB$STOP require termination of the operation that raises the condition. The condition handler will not be allowed to continue from the point of call to LIB$STOP.

- Display an informational message

  The system establishes default condition handlers before it calls the main program. Because these default condition handlers provide access to the system's standard error messages, the standard method for displaying a message is by signaling the severity of the condition: informational, warning, or error. See Section 13.5 for the definition of the severity field of a condition vector. The system default condition handlers resume execution of the instruction after displaying the messages associated with the signal. If the condition value indicates a severe condition, then the image exits after the message is displayed.

- Display a stack traceback on errors

  The default operations of the LINK and RUN commands provide a system-supplied handler (the traceback handler) to print a symbolic stack traceback. The traceback shows the state of the routine stack at the point where the condition occurred. The traceback information is displayed along with the messages associated with the signaled condition.

- Compile customer-defined messages

  The Message utility allows you to define your own exception conditions and the associated messages. Message source files contain the condition values and their associated messages. See Section 13.11.3 for a complete description of how to define your own messages.

- Unwind the stack

  A condition handler can cause a signal to be dismissed and the stack to be unwound to the establisher or caller of the establisher of the condition handler when it returns control to the OpenVMS Condition Handling facility (CHF). During the unwinding operation, the CHF scans the stack. If a condition handler is associated with a frame, the system calls that handler before removing the frame. Calling the condition handlers during the unwind allows a routine to perform cleanup operations specific to a particular application, such as recovering from noncontinuable errors or deallocating resources that were allocated by the routine (such as virtual memory, event flags, and so forth). See Section 13.12.3 for a description of the SYS$UNWIND system service.

- Log error messages to a file

  The Put Message (SYS$PUTMSG) system service permits any user-written handler to include a message in a listing file. Such message logging can be separate from the default messages the user receives. See Section 13.11 for a detailed description of the SYS$PUTMSG system service.

13.3 Exception Conditions

Exceptions can be generated by any of the following:

- Hardware
- Software
- System service failures

Hardware-generated exceptions always result in conditions that require special action if program execution is to continue.

Software-generated exceptions may result in error or warning conditions. These conditions and their message descriptions are documented in the online Help Message utility and in the OpenVMS system messages documentation. To access online message descriptions, use the HELP/MESSAGE command.

More information on using the Help Message utility is available in *OpenVMS System Messages: Companion Guide for Help Message Users*. That document describes only those messages that occur when the system is not fully operational and you cannot access Help Message.

Some examples of exception conditions are as follows:

- Arithmetic exception condition in a user-written program detected and signaled by hardware (for example, floating-point overflow)
- Error in a user argument to a run-time library routine detected by software and signaled by calling LIB$STOP (for example, a negative square root)
- Error in a run-time library language-support routine, such as an I/O error or an error in a data-type conversion
- RMS success condition stating that the record is already locked
- RMS success condition stating that the created file superseded an existing version

There are two standard methods for a Digital- or user-written routine to indicate that an exception condition has occurred:

- Return a completion code to the calling program using the function value mechanism

  Most general-purpose run-time library routines indicate exception conditions by returning a condition value in R0. The calling program then tests bit 0 of R0 for success or failure. This method allows better programming structure, because the flow of control can be changed explicitly after the return from each call. If the actual function value returned is greater than 32 bits, then use both R0 and R1.

  Alpha On Alpha systems, if the actual function returned is a floating-point value, the floating-point value is returned in F0, or F0 and F1. ♦

- Signal the exception condition

  A condition can be signaled by calling the RTL routine LIB$SIGNAL or LIB$STOP. Any condition handlers that were enabled are then called by the CHF. See Figure 13–5 for the order in which CHF invokes condition handlers.

  Exception conditions raised by hardware or software are signaled to the routine identically.

  For more details, see Section 13.8 and Section 13.8.1.

13.3.1 System Service Exception Conditions

System service failure exceptions occur when an error or severe error status is returned from a call to a system service. You can choose to handle error returns from system services by using the condition-handling mechanism rather than other error-checking methods. If you want to handle exceptions generated by service failures, you must enable system service failure exception mode with the Set System Service Failure Mode (SYS$SETSFM) system service. For example:

```
$SETSFM_S ENBFLG=#1
```

System service failure exception mode is initially disabled, and it can be enabled or disabled at any time during the execution of an image.

13.3.1.1 Conditions Caused by Exceptions

Table 13–1 summarizes common conditions caused by exceptions. The condition names are listed in the first column. The second column explains each condition more fully by giving information about the type, meaning, and arguments relating to the condition. The condition type is either trap or fault. For more information about traps and faults, refer to the *VAX Architecture Reference Manual* and *Alpha Architecture Reference Manual*. The meaning of the exception condition is a short description of each condition. The arguments for the condition handler are listed where applicable; they give specific information about the condition.

Table 13–1 Summary of Exception Conditions

Condition Name	Explanation	
SS$_ACCVIO	Type:	Fault.
	Description:	Access violation.
	Arguments:	1. Reason for access violation. This is a mask with the following format: Bit <0> = type of access violation Bit <0> = page table entry protection code did not permit intended access † Bit <1> = P0LR, P1LR, or SLR length violation Bit <1> = page table entry reference Bit <0> = specified virtual address not accessible Bit <1> = associated page table entry not accessible Bit <2> = intended access Bit <0> = read Bit <1> = modify 2. Virtual address to which access was attempted or, on some processors, virtual address within the page to which access was attempted.
†SS$_ARTRES	Type:	Trap.
	Description:	Reserved arithmetic trap.
	Arguments:	None.

†On VAX systems, this condition is generated by hardware.

(continued on next page)

Table 13–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS$_ASTFLT	Type:	Trap.
	Description:	Stack invalid during attempt to deliver an AST.
	Arguments:	1. Stack pointer value when fault occurred. 2. AST parameter of failed AST. 3. Program counter (PC) at AST delivery interrupt. 4. Processor status longword (PSL) for VAX or processor status (PS) for Alpha at AST delivery interrupt.[1] For PS, it is the low-order 32 bits. 5. Program counter (PC) to which AST would have been delivered.[1] 6. Processor status longword (PSL) for VAX or processor status (PS) for Alpha to which AST would have been delivered.[1] For PS, it is the low-order 32 bits.
SS$_BREAK	Type:	Fault.
	Description:	Breakpoint instruction encountered.
	Arguments:	None.
SS$_CMODSUPR	Type:	Trap.
	Description:	Change mode to supervisor instruction encountered.[2]
	Arguments:	Change mode code. The possible values are –32,768 through 32,767.
SS$_CMODUSER	Type:	Trap.
	Description:	Change mode to user instruction encountered.[2]
	Arguments:	Change mode code. The possible values are –32,768 through 32,767.

[1]The PC and PSL (or PS) normally included in the signal array are not included in this argument list. The stack pointer of the access mode receiving this exception is reset to its initial value.

[2]If a change mode handler has been declared for user or supervisor mode with the Declare Change Mode or Compatibility Mode Handler (SYS$DCLCMH) system service, that routine receives control when the associated trap occurs.

(continued on next page)

Table 13–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
†SS$_COMPAT	Type:	Fault.
	Description:	Compatibility-mode exception. This exception condition can occur only when executing in compatibility mode.[3]
	Arguments:	Type of compatibility exception. The possible values are as follows: 0 = Reserved instruction execution 1 = BPT instruction executed 2 = IOT instruction executed 3 = EMT instruction executed 4 = TRAP instruction executed 5 = Illegal instruction executed 6 = Odd address fault 7 = TBIT trap.
†‡SS$_DECOVF	Type:	Trap.
	Description:	Decimal overflow.
	Arguments:	None.
†‡SS$_FLTDIV	Type:	Trap.
	Description:	Floating/decimal divide-by-zero.
	Arguments:	None.
†SS$_FLTDIV_F	Type:	Fault.
	Description:	Floating divide-by-zero.
	Arguments:	None.
†‡SS$_FLTOVF	Type:	Trap.
	Description:	Floating-point overflow.
	Arguments:	None.
†SS$_FLTOVF_F	Type:	Fault.
	Description:	Floating-point overflow fault.
	Arguments:	None.
†‡SS$_FLTUND	Type:	Trap.
	Description:	Floating-point underflow.
	Arguments:	None.
†SS$_FLTUND_F	Type:	Fault.
	Description:	Floating-point underflow fault.
	Arguments:	None.
†‡SS$_INTDIV	Type:	Trap.
	Description:	Integer divide-by-zero.
	Arguments:	None.

[3]If a compatibility-mode handler has been declared with the Declare Change Mode or Compatibility Mode Handler (SYS$DCLCMH) system service, that routine receives control when this fault occurs.

†On VAX systems, this condition is generated by hardware.

‡On Alpha systems, this condition is generated by software.

(continued on next page)

Table 13–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
†‡SS$_INTOVF	Type:	Trap.
	Description:	Integer overflow.
	Arguments:	None.
†SS$_OPCCUS	Type:	Fault.
	Description:	Opcode reserved for customer fault.
	Arguments:	None.
SS$_OPCDEC	Type:	Fault.
	Description:	Opcode reserved for Digital fault.
	Arguments:	None.
SS$_PAGRDERR	Type:	Fault.
	Description:	Read error occurred during an attempt to read a faulted page from disk.
	Arguments:	1. Translation not valid reason. This is a mask with the following format: Bit <0> = 0 Bit <1> = page table entry reference Bit <0> = specified virtual address not valid Bit <1> = associated page table entry not valid Bit <2> = intended access Bit <0> = read Bit <1> = modify 2. Virtual address of referenced page.
†SS$_RADRMOD	Type:	Fault.
	Description:	Attempt to use a reserved addressing mode.
	Arguments:	None.
SS$_ROPRAND	Type:	Fault.
	Description:	Attempt to use a reserved operand.
	Arguments:	None.
SS$_SSFAIL	Type:	Fault.
	Description:	System service failure (when system service failure exception mode is enabled).
	Arguments:	Status return from system service (R0). (The same value is in R0 of the mechanism array.)
†‡SS$_SUBRNG	Type:	Trap.
	Description:	Subscript range trap.
	Arguments:	None.

†On VAX systems, this condition is generated by hardware.
‡On Alpha systems, this condition is generated by software.

(continued on next page)

Table 13–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
†SS$_TBIT	Type:	Fault.
	Description:	Trace bit is pending following an instruction.
	Arguments:	None.

†On VAX systems, this condition is generated by hardware.

Change-Mode and Compatibility-Mode Handlers

Two types of hardware exception can be handled in a way different from the normal condition-handling mechanism described in this chapter. The two types of hardware exception are as follows:

VAX

- Traps caused by change-mode-to-user or change-mode-to-supervisor instructions
- Compatibility mode faults ♦

You can use the Declare Change Mode or Compatibility Mode Handler (SYS$DCLCMH) system service to establish procedures to receive control when one of these conditions occurs. The SYS$DCLCMH system service is described in the *OpenVMS System Services Reference Manual*.

13.3.2 Exception Conditions

Alpha

On Alpha systems, the condition values that your condition-handling routine expects to receive on VAX systems may no longer be meaningful, even though the format of the 32-bit condition value and its location in the signal array are the same as they are on VAX systems. Because of architectural differences, some exception conditions that are returned on VAX systems are not supported on Alpha systems.

Because hardware exceptions are more architecture specific than software exceptions, only a subset of the hardware exceptions supported on VAX systems are also supported on Alpha systems. In addition, the Alpha architecture defines several additional exceptions that are not supported on VAX systems. ♦

Table 13–2 lists the Alpha exceptions that are not supported on VAX systems and VAX hardware exceptions that are not supported on Alpha systems.

Table 13–2 Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
New Alpha Exceptions	
SS$_HPARITH–High-performance arithmetic exception	Generated for all Alpha arithmetic exceptions (see Section 13.3.3)
SS$_ALIGN–Data alignment trap	No VAX equivalent

Exception Condition Code	Comment
VAX-Specific Hardware Exceptions	
SS$_ARTRES–Reserved arithmetic trap	No Alpha system equivalent
SS$_COMPAT–Compatibility fault	No Alpha system equivalent
‡SS$_DECOVF–Decimal overflow	Replaced by SS$_HPARITH (see Section 13.3.3)
‡SS$_FLTDIV–Float divide-by-zero (trap)	Replaced by SS$_HPARITH (see Section 13.3.3)
SS$_FLTDIV_F–Float divide-by-zero (fault)	Replaced by SS$_HPARITH (see Section 13.3.3)
‡SS$_FLTOVF–Float overflow (trap)	Replaced by SS$_HPARITH (see Section 13.3.3)
SS$_FLTOVF_F–Float overflow (fault)	Replaced by SS$_HPARITH (see Section 13.3.3)
‡SS$_FLTUND–Float underflow (trap)	Replaced by SS$_HPARITH (see Section 13.3.3)
SS$_FLTUND_F–Float underflow (fault)	Replaced by SS$_HPARITH (see Section 13.3.3)
‡SS$_INTDIV–Integer divide-by-zero	Replaced by SS$_HPARITH (see Section 13.3.3)
‡SS$_INTOVF–Integer overflow	Replaced by SS$_HPARITH (see Section 13.3.3)
SS$_TBIT–Trace pending	No Alpha system equivalent
SS$_OPCCUS–Opcode reserved to customer	No Alpha system equivalent
SS$_RADMOD–Reserved addressing mode	No Alpha system equivalent
SS$_SUBRNG–INDEX subscript range check	No Alpha system equivalent

‡On Alpha systems, this condition may be generated by software.

13.3.3 Arithmetic Exceptions

VAX

On VAX systems, the architecture ensures that arithmetic exceptions are reported synchronously; that is, a VAX arithmetic instruction that causes an exception (such as an overflow) enters any exception handlers immediately, and subsequent instructions are not executed. The program counter (PC) reported to the exception handler is that of the failing arithmetic instruction. This allows application programs, for example, to resume the main sequence, with the failing operation being emulated or replaced by some equivalent or alternative set of operations. ♦

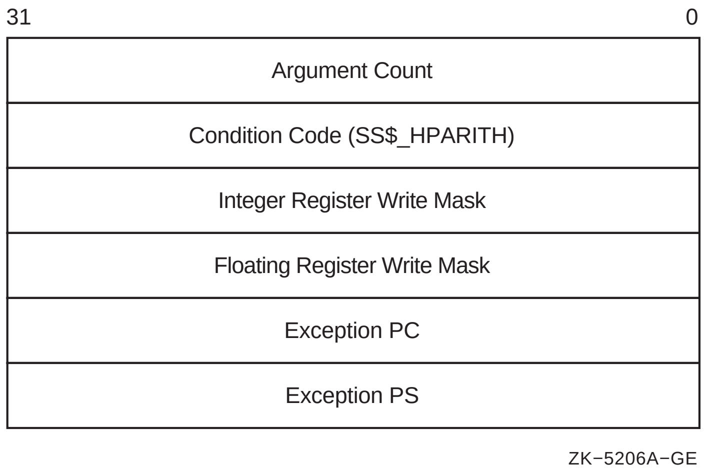
Alpha

On Alpha systems, arithmetic exceptions are reported **asynchronously**; that is, implementations of the architecture can allow a number of instructions (including branches and jumps) to execute beyond that which caused the exception. These instructions may overwrite the original operands used by the failing instruction, thus causing the loss of information that is integral to interpreting or rectifying the exception. The program counter (PC) reported to the exception handler is not that of the failing instruction, but rather is that of some subsequent instruction. When the exception is reported to an application's exception handler, it may be impossible for the handler to fix up the input data and restart the instruction.

Because of this fundamental difference in arithmetic exception reporting, Alpha systems define a new, single condition code, SS$_HPARITH, to indicate all arithmetic exceptions. Thus, if your application contains a condition-handling routine that performs processing when an integer overflow exception occurs, on VAX systems the application expects to receive the SS$_INTOVF condition code. On Alpha systems, this exception is indicated by the condition code SS$_HPARITH. In this way, condition-handling routines in VAX applications cannot mistake an Alpha system arithmetic exception for the corresponding VAX exception. This is important because the processing performed by the VAX application assumes that the exception is reported synchronously, which is not the case on Alpha systems.

Figure 13–1 shows the format of the SS$_HPARITH exception signal array.

Figure 13–1 SS$_HPARITH Exception Signal Array


31
0
Argument Count
Condition Code (SS$_HPARITH)
Integer Register Write Mask
Floating Register Write Mask
Exception PC
Exception PS
ZK−5206A−GE

This signal array contains three arguments that are specific to the SS$_HPARITH exception: the **integer register write mask**, **floating register write mask**, and **exception summary** arguments of the **exception pc** and **exception ps**. The **integer register write mask** and **floating register write mask** arguments indicate the registers that were targets of instructions that set bits in the **exception summary** argument. Each bit in the mask represents a register. The **exception summary** argument indicates the type of exceptions that are being signaled by setting flags in the first 7 bits. Table 13–3 lists the meaning of each of these bits when set.

Table 13–3 Exception Summary Argument Fields

Bit	Meaning When Set
0	Software completion.
1	Invalid floating arithmetic, conversion, or comparison operation.
2	Invalid attempt to perform a floating divide operation with a divisor of zero. Note that integer divide-by-zero is not reported.
3	Floating arithmetic or conversion operation overflowed the destination exponent.
4	Floating arithmetic or conversion operation underflowed the destination exponent.
5	Floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
6	Integer arithmetic or conversion operation from floating point to integer overflowed the destination precision.

For more information and recommendations about using arithmetic exceptions on Alpha systems, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*. ♦

13.3.4 Unaligned Access Traps (Alpha Only)

Alpha

On Alpha systems, an unaligned access trap is generated when an attempt is made to load or store a longword or quadword to or from a register using an address that does not have the natural alignment of the particular data reference and not using an Alpha instruction that takes an unaligned address as an operand (LDQ_U). For more information about data alignment, see Section 13.4.2.

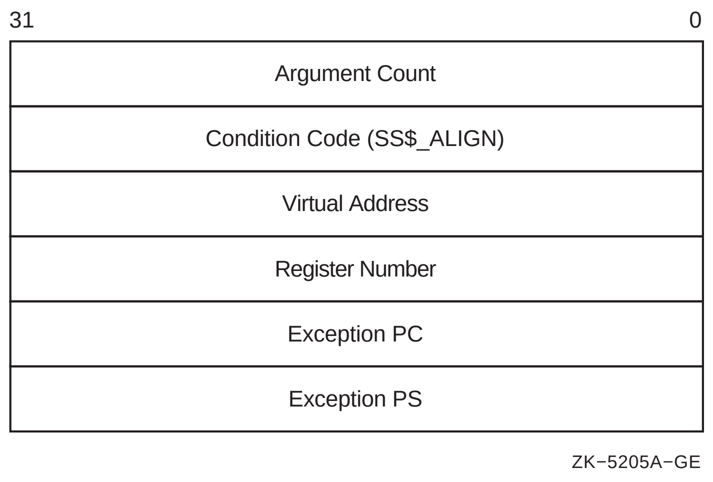
Alpha compilers typically avoid triggering alignment faults by:

- Aligning static data on natural boundaries by default. (This default behavior can be overridden by using a compiler qualifier.)
- Generating special inline code sequences for data that is known to be unnaturally aligned at compile time.

Note, however, that compilers cannot align dynamically defined data. Thus, alignment faults may be triggered.

An alignment exception is identified by the condition code SS$_ALIGN. Figure 13–2 illustrates the elements of the signal array returned by the SS$_ALIGN exception.

Figure 13–2 SS$_ALIGN Exception Signal Array



This signal array contains two arguments specific to the SS$_ALIGN exception: the **virtual address** argument and the **register number** argument. The **virtual address** argument contains the address of the unaligned data being accessed. The **register number** argument identifies the target register of the operation. ♦

13.4 How Run-Time Library Routines Handle Exceptions

Most general-purpose run-time library routines handle errors by returning a status in R0. In some cases, however, exceptions that occur during the execution of a run-time library routine are signaled. This section tells how run-time library routines signal exception conditions.

Some calls to the run-time library do not or cannot specify an action to be taken. In this case, the run-time library signals the proper exception condition by using the operating system's signaling mechanism.

In order to maintain modularity, the run-time library does not use exception vectors, which are processwide data locations. Thus, the run-time library itself does not establish handlers by using the primary, secondary, or last-chance exception vectors.

13.4.1 Exception Conditions Signaled from Mathematics Routines (VAX Only)

VAX

On VAX systems, mathematics routines return function values in register R0 or registers R0 and R1, unless the return values are larger than 64 bits. For this reason, mathematics routines cannot use R0 to return a completion status and must signal all errors. In addition, all mathematics routines signal an error specific to the MTH$ facility rather than a general hardware error.

13.4.1.1 Integer Overflow and Floating-Point Overflow

Although the hardware normally detects integer overflow and floating-point overflow errors, run-time library mathematics routines are programmed with a software check to trap these conditions before the hardware signaling process can occur. This means that they call LIB$SIGNAL instead of allowing the hardware to initiate signaling.

The software check is needed because JSB routines cannot set up condition handlers. The check permits the JSB mathematics routines to add an extra stack frame so that the error message and stack traceback appear as if a CALL instruction had been performed. Because of the software check, JSB routines do not cause a hardware exception condition even when the calling program has enabled the detection of integer overflow. On the other hand, detection of floating-point overflow is always enabled and cannot be disabled.

If an integer or floating-point overflow occurs during a CALL or a JSB routine, the routine signals a mathematics-specific error such as MTH$_FLOOVEMAT (Floating Overflow in Math Library) by calling LIB$SIGNAL explicitly.

13.4.1.2 Floating-Point Underflow

All mathematics routines are programmed to avoid floating-point underflow conditions. Software checks are made to determine if a floating-point underflow condition would occur. If so, the software makes an additional check:

- If the immediate calling program (CALL or JSB) has enabled floating-point underflow traps, a mathematics-specific error condition is signaled.
- Otherwise, the result is corrected to zero and execution continues with no error condition.

The user can enable or disable detection of floating-point underflow at run time by calling the routine LIB$FLT_UNDER. ♦

13.4.2 System-Defined Arithmetic Condition Handlers

VAX

On VAX systems, you can use the following run-time library routines as arithmetic condition handlers to enable or disable the signaling of decimal overflow, floating-point underflow, and integer overflow:

- LIB$DEC_OVER—Enables or disables the signaling of a decimal overflow. By default, signaling is disabled.
- LIB$FLT_UNDER—Enables or disables the signaling of a floating-point underflow. By default, signaling is disabled.
- LIB$INT_OVER—Enables or disables the signaling of an integer overflow. By default, signaling is enabled.

You can establish these handlers in one of two ways:

- Invoke the appropriate handler as a function specifying the first argument as 1 to enable signaling.
- Invoke the handler with command qualifiers when you compile your program. (Refer to your program language manuals.)

You cannot disable the signaling of integer divide-by-zero, floating-point overflow, and floating-point or decimal divide-by-zero.

When the signaling of a hardware condition is enabled, the occurrence of the exception condition causes the operating system to signal the condition as a severe error. When the signaling of a hardware condition is disabled, the occurrence of the condition is ignored, and the processor executes the next instruction in the sequence.

The signaling of overflow and underflow detection is enabled independently for activation of each routine, because the call instruction saves the state of the calling program's hardware enable operations in the stack and then initializes the enable operations for the called routine. A return instruction restores the calling program's enable operations.

These run-time library routines are intended primarily for high-level languages, because you can achieve the same effect in MACRO with the single Bit Set PSW (BISPSW) or Bit Clear PSW (BICPSW) VAX instructions.

These routines allow you to enable and disable detection of decimal overflow, floating-point underflow, and integer overflow for a portion of your routine's execution. Note that the VAX BASIC and DEC Fortran compilers provide a compile-time qualifier that permits you to enable or disable integer overflow for your entire routine. ♦

Alpha

On Alpha systems, certain RTL routines that process conditions do not exist because the exception conditions defined by the Alpha architecture differ somewhat from those defined by the VAX architecture. Table 13–4 lists the run-time library condition-handling support routines available on VAX systems and indicates which are supported on Alpha systems. ♦

Table 13–4 Run-Time Library Condition-Handling Support Routines

Routine	Availability on Alpha Systems
Arithmetic Exception Support Routines	
LIB$DEC_OVER–Enables or disables signaling of decimal overflow	Not supported
LIB$FIXUP_FLT–Changes floating-point reserved operand to a specified value	Not supported
LIB$FLT_UNDER–Enables or disables signaling of floating-point underflow	Not supported
LIB$INT_OVER–Enables or disables signaling of integer overflow	Not supported
General Condition-Handling Support Routines	
LIB$DECODE_FAULT–Analyzes instruction context for fault	Not supported
LIB$ESTABLISH–Establishes a condition handler	Not supported (languages may support for compatibility)
LIB$MATCH_COND–Matches condition value	Supported
LIB$REVERT-Deletes a condition handler	Not supported (languages may support for compatibility)
LIB$SIG_TO_STOP–Converts a signaled condition to a condition that cannot be continued	Supported
LIB$SIG_TO_RET–Converts a signal to a return status	Supported
LIB$SIM_TRAP–Simulates a floating-point trap	Not supported
LIB$SIGNAL–Signals an exception condition	Supported
LIB$STOP–Stops execution by using signaling	Supported

13.5 Condition Values

Error conditions are identified by integer values called **condition codes** or **condition values**. The operating system defines condition values to identify errors that might occur during execution of system-defined procedures. Each exception condition has associated with it a unique, 32-bit condition value that identifies the exception condition, and each condition value has a unique, systemwide symbol and an associated message. The condition value is used in both methods of indicating exception conditions, returning a status and signaling.

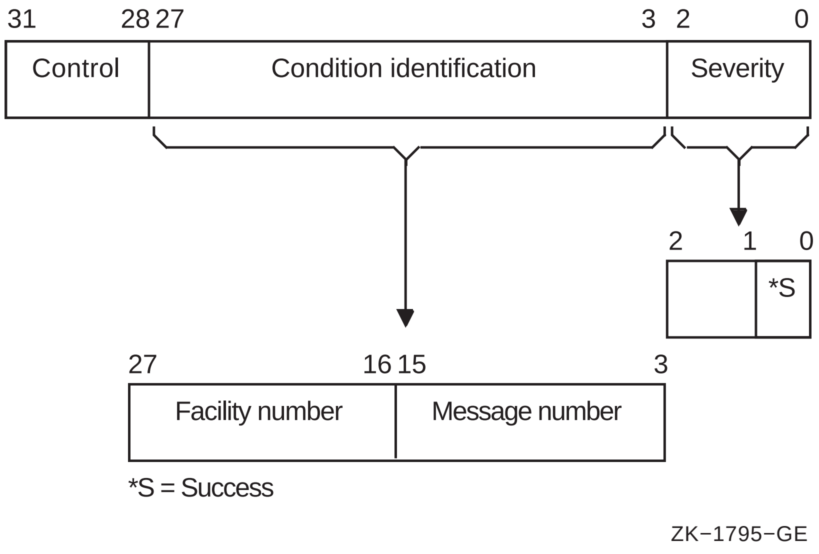
From a condition value you can determine whether an error has occurred, which error has occurred, and the severity of the error. Table 13–5 describes the fields of a condition value.

Table 13–5 Fields of a Condition Value

Field	Bits	Meaning
FAC_NO	<27:16>	Indicates the system facility in which the condition occurred
MSG_NO	<15:3>	Indicates the condition that occurred
SEVERITY	<2:0>	Indicates whether the condition is a success (bit <0> = 1) or a failure (bit <0> = 0) as well as the severity of the error, if any

Figure 13–3 shows the format of a condition value.

Figure 13–3 Format of a Condition Value



Condition Value Fields

severity

The severity of the error condition. Bit <0> indicates success (logical true) when set and failure (logical false) when clear. Bits <1> and <2> distinguish degrees of success or failure. The three bits, when taken as an unsigned integer, are interpreted as described in Table 13–6. The symbolic names are defined in module $STSDEF.

Table 13–6 Severity of Error Conditions

Value	Symbol	Severity	Response
0	STS$K_WARNING	Warning	Execution continues, unpredictable results
1	STS$K_SUCCESS	Success	Execution continues, expected results
2	STS$K_ERROR	Error	Execution continues, erroneous results
3	STS$K_INFO	Information	Execution continues, informational message displayed
4	STS$K_SEVERE	Severe error	Execution terminates, no output
5			Reserved for Digital
6			Reserved for Digital
7			Reserved for Digital

condition identification
Identifies the condition uniquely on a systemwide basis.

control
Four control bits. Bit <28> inhibits the message associated with the condition value from being printed by the SYS$EXIT system service. After using the SYS$PUTMSG system service to display an error message, the system default handler sets this bit. It is also set in the condition value returned by a routine as a function value, if the routine has also signaled the condition, so that the condition has been either printed or suppressed. Bits <29:31> must be zero; they are reserved for Digital.

When a software component completes execution, it returns a condition value in this format. When a severity value of warning, error, or severe error has been generated, the status value returned describes the nature of the problem. Your program can test this value to change the flow of control or to generate a message. Your program can also generate condition values to be examined by other routines and by the command language interpreter. Condition values defined by customers must set bits <27> and <15> so that these values do not conflict with values defined by Digital.

message number
The number identifying the message associated with the error condition. It is a status identification, that is, a description of the hardware exception condition that occurred or a software-defined value. Message numbers with bit <15> set are specific to a single facility. Message numbers with bit <15> clear are systemwide status values.

facility number
Identifies the software component generating the condition value. Bit <27> is set for user facilities and clear for Digital facilities.

13.5.1 Return Status Convention

Most system-defined procedures are functions of longwords, where the function value is equated to a condition value. In this capacity, the condition value is referred to as a **return status**. You can write your own routines to follow this convention. See Section 13.14.2 for information about how to change a signal to a return status. Each routine description in the *OpenVMS System Services Reference Manual*, *OpenVMS RTL Library (LIB$) Manual*, *OpenVMS Record Management Utilities Reference Manual*, and *OpenVMS Utility Routines Manual* lists the condition values that can be returned by that procedure.

13.5.1.1 Testing Returned Condition Values

When a function returns a condition value to your program unit, you should always examine the returned condition value. To check for a failure condition (warning, error, or severe error), test the returned condition value for a logical value of false. The following program segment invokes the run-time library procedure LIB$DATE_TIME, checks the returned condition value (returned in the variable STATUS), and, if an error has occurred, signals the condition value by calling the run-time library procedure LIB$SIGNAL (Section 13.8 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 value. For example, LIB$DATE_TIME returns a success value (LIB$_STRTRU) when it truncates the string. If you want to take special action when truncation occurs, specify the condition as shown in the following example (the special action would follow the IF statement):

```
INTEGER*4 STATUS,
2         LIB$DATE_TIME
CHARACTER*23 DATE

INCLUDE '($LIBDEF)'
   .
   .
   .
STATUS = LIB$DATE_TIME (DATE)
IF (STATUS .EQ. LIB$_STRTRU) THEN
   .
   .
   .
```

13.5.1.2 Testing SS$_NOPRIV and SS$_EXQUOTA Condition Values

The SS$_NOPRIV and SS$_EXQUOTA condition values returned by a number of system service procedures require special checking. Any system service that is listed as returning SS$_NOPRIV or SS$_EXQUOTA can instead return a more specific condition value that indicates the privilege or quota in question. Table 13–7 list the specific privilege errors, and Table 13–8 lists the quota errors.

Table 13–7 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		

Table 13–8 Quota Errors

SS$_EXASTLM	SS$_EXBIOLM	SS$_EXBYTLM
SS$_EXDIOLM	SS$_EXENQLM	SS$_EXFILLM
SS$_EXPGFLQUOTA	SS$_EXPRCLM	SS$_EXTQELM

Because either a general or a specific value can be returned, your program must test for both. The following four symbols provide a starting and ending point with which you can compare the returned condition value:

- SS$_NOPRIVSTRT—First specific value for SS$_NOPRIV
- SS$_NOPRIVEND—Last specific value for SS$_NOPRIV
- SS$_NOQUOTASTRT—First specific value for SS$_EXQUOTA
- SS$_NOQUOTAEND—Last specific value for SS$_EXQUOTA

The following DEC Fortran example tests for a privilege error by comparing STATUS (the returned condition value) with the specific condition value 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
```

13.5.2 Modifying Condition Values

To modify a condition value, copy a series of bits from one longword to another longword. For example, the following statement copies the first three bits (bits <2:0>) of STS$K_INFO to the first three bits of the signaled condition code, which is in the second element of the signal array named SIGARGS. As shown in Table 13–6, STS$K_INFO contains the symbolic severity code for an informational message.

```
! Declare STS$K_ symbols
INCLUDE '($STSDEF)'
   .
   .
   .
! Change the severity of the condition code
! in SIGARGS(2) to informational
CALL MVBITS (STS$K_INFO,
2            0,
2            3,
2            SIGARGS(2),
2            0)
```

Once you modify the condition value, you can resignal the condition value and either let the default condition handler display the associated message or use the SYS$PUTMSG system service to display the message. If your condition handler displays the message, do not resignal the condition value, or the default condition handler will display the message a second time.

In the following example, the condition handler verifies that the signaled condition value is LIB$_NOSUCHSYM. If it is, the handler changes its severity from error to informational and then resignals the modified condition value. 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
```

13.6 Exception Dispatcher

When an exception occurs, control is passed to the operating system's exception-dispatching routine. The exception dispatcher searches for a condition-handling routine invoking the first handler it finds and passes the information to the handler about the condition code and the state of the program when the condition code was signaled. If the handler resignals, the operating system searches for another handler; otherwise, the search for a condition handler ends.

The operating system searches for condition handlers in the following sequence:

1. Primary exception vectors—Four vectors (lists) of one or more condition handlers; each vector is associated with an access mode. By default, all of the primary exception vectors are empty. Exception vectors are used primarily for system programming, not application programming. The debugger uses the primary exception vector associated with user mode.

   When an exception occurs, the 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 that are entered into the exception vectors associated with kernel, executive, and supervisor modes remain in effect until they are canceled or until you log out. Condition handlers that are entered into the exception vector associated with user mode remain in effect until they are canceled or until the image that entered them exits.

2. Secondary exception vectors—A set of exception vectors with the same structure as the primary exception vectors. Exception vectors are primarily used for system programming, not application programming. By default, all of the secondary exception vectors are empty.

3. Call frame condition handlers—Each program unit can establish one condition handler (the address of the handler is placed in the call frame of the program unit). The 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 operating system searches for a handler that was established by the program unit that invoked the current program unit, and so on back to the main program.

4. Traceback handler—If you do not establish any condition handlers and link your program with the /TRACEBACK qualifier of the LINK command (the default), the operating system finds and invokes the traceback handler.
5. Catchall handler—If you do not establish any condition handlers and you link your program with the /NOTRACEBACK qualifier to the LINK command, the operating system finds and invokes the catchall handler. The catchall handler is at the bottom of the user stack and in the last-chance exception vector.
6. Last-chance exception vectors—A set of exception vectors with the same structure as the primary and secondary exception vectors. Exception vectors are used primarily 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.

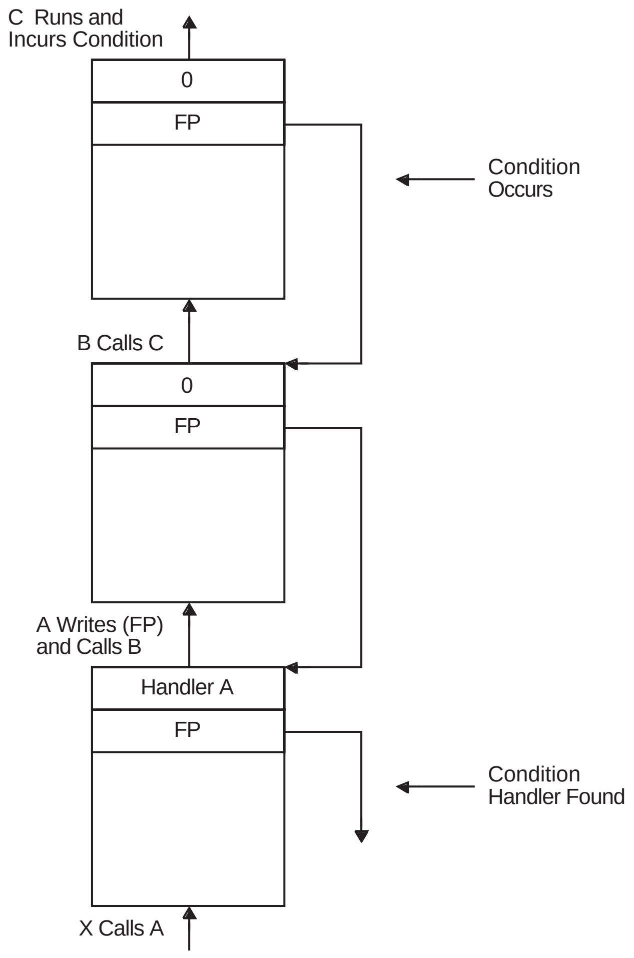
The search is terminated when the dispatcher finds a condition handler. If the dispatcher cannot find a user-specified condition handler, it calls the condition handler whose address is stored in the last-chance exception vector. If the image was activated by the command language interpreter, the last-chance vector points to the catchall condition handler. The catchall handler issues a message and either continues program execution or causes the image to exit, depending on whether the condition was a warning or an error condition, respectively.

You can call the catchall handler in two ways:

- If the last-chance exception vector returns to the dispatcher or if the last-chance exception vector is empty, the last-chance exception vector calls the catchall condition handler and exits with the return status code SS$_NOHANDLER.
- If the exception dispatcher detects an access violation, it calls the catchall condition handler and exits with the return status code SS$_ACCVIO.

Figure 13–4 illustrates the exception dispatcher's search of the call stack for a condition handler.

Figure 13–4 Searching the Stack for a Condition Handler



1 The illustration of the call stack indicates the calling sequence: Procedure A calls procedure B, and procedure B calls procedure C. Procedure A establishes a condition handler.

2 An exception occurs while procedure C is executing. The exception dispatcher searches for a condition handler.

3 After checking for a condition handler declared in the exception vectors (assume that none has been specified for the process), the dispatcher looks at the first longword of procedure C's call frame. A value of 0 indicates that no condition handler has been specified. The dispatcher locates the call frame for procedure B by using the frame pointer (FP) in procedure C's call frame. Again, it finds no condition handler, and locates procedure A's call frame.

4 The dispatcher locates and gives control to handler A.

ZK−0858−GE

In cases where the default condition handling is insufficient, you can establish your own handler by one of the mechanisms described in Section 13.2.1. Typically, you need condition handlers only if your program must perform one of the following operations:

- Respond to condition values that are signaled rather than returned, such as an integer overflow error. (Section 13.14.2 describes the system-defined handler LIB$SIG_TO_RET that allows you to treat signals as return values; Section 13.4.2 describes other useful system-defined handlers for arithmetic errors.)
- Modify part of a condition code, such as the severity. See Section 13.5.2 for more information. If you want to change the severity of any condition code to a severe error, you can use the run-time library procedure LIB$STOP instead of writing your own condition handler.
- Add messages to the one associated with the originally signaled condition code or log the messages associated with the originally signaled condition code.

13.7 Argument List Passed to a Condition Handler

VAX

On VAX systems, the argument list passed to the condition handler is constructed on the stack and consists of the addresses of two argument arrays, signal and mechanism, as illustrated in Section 13.8.2 and Section 13.8.3. ♦

Alpha

On Alpha systems, the arrays are set up on the stack, but any argument is passed in registers. ♦

VAX

On VAX systems, you can use the $CHFDEF macro instruction to define the symbolic names to refer to the arguments listed in Table 13–9.

Table 13–9 $CHFDEF Symbolic Names and Arguments on VAX Systems

Symbolic Name	Related Argument
CHF$L_SIGARGLST	Address of signal array
CHF$L_MCHARGLST	Address of mechanism array
CHF$L_SIG_ARGS	Number of signal arguments
CHF$L_SIG_NAME	Condition name
CHF$L_SIG_ARG1	First signal-specific argument
CHF$L_MCH_ARGS	Number of mechanism arguments
CHF$L_MCH_FRAME	Establisher frame address
CHF$L_MCH_DEPTH	Frame depth of establisher
CHF$L_MCH_SAVR0	Saved register R0
CHF$L_MCH_SAVR1	Saved register R1 ♦

Alpha

On Alpha systems, you can use the $CHFDEF2 macro instruction to define the symbolic names to refer to the arguments listed in Table 13–10.

Table 13–10 $CHFDEF2 Symbolic Names and Arguments on Alpha Systems

Symbolic Name	Related Argument
CHF$L_SIGARGLST	Address of signal array
CHF$L_MCHARGLST	Address of mechanism array
CHF$IS_SIG_ARGS	Number of signal arguments
CHF$IS_SIG_NAME	Condition name
CHF$IS_SIG_ARG1	First signal-specific argument
CHF$IS_MCH_ARGS	Number of mechanism arguments
CHF$IS_MCH_FLAGS	Flag bits <63:0> for related argument mechanism information
CHF$PH_MCH_FRAME	Establisher frame address
CHF$IS_MCH_DEPTH	Frame depth of establisher
CHF$PH_MCH_DADDR	Address of the handler data quadword if the exception handler data field is present
CHF$PH_MCH_ESF_ADDR	Address of the exception stack frame
CHF$PH_MCH_SIG_ADDR	Address of the signal array
CHF$IH_MCH_SAVR*nn*	Contains a copy of the saved integer registers at the time of the exception
CHF$FH_MCH_SAVF*nn*	Contains a copy of the saved floating-point registers at the time of the exception ♦

13.8 Signaling

Signaling can be initiated when hardware or software detects an exception condition. In either case, the exception condition is said to be signaled by the routine in which it occurred. If hardware detects the error, it passes control to a condition dispatcher. If software detects the error, it calls one of the run-time library signal-generating routines: LIB$SIGNAL or LIB$STOP. The RTL signal-generating routines pass control to the same condition dispatcher. When LIB$STOP is called, the severity code is forced to severe, and control cannot return to the routine that signaled the condition. See Section 13.12.1 for a description of how a signal can be dismissed and how normal execution from the point of the exception condition can be continued.

When a routine signals, it passes to the OpenVMS Condition Handling facility (CHF) the condition value associated with the exception condition, as well as optional arguments that can be passed to a condition handler. The CHF uses these arguments to build two data structures on the stack:

- The signal argument vector. This vector contains the information describing the nature of the exception condition.
- The mechanism argument vector. This vector describes the state of the process at the time the exception condition occurred.

These two vectors become the arguments that the CHF passes to condition handlers.

These argument vectors are described in detail in Section 13.8.2 and Section 13.8.3.

After the signal and mechanism argument vectors are set up, the CHF searches for enabled **condition handlers**. A condition handler is a separate routine that has been associated with a routine in order to take a specific action when an exception condition occurs. The CHF searches for condition handlers to handle the exception condition, beginning with the primary exception vector of the access mode in which the exception condition occurred. If this vector contains the address of a handler, that handler is called. If the address is 0 or if the handler resignals, then the CHF repeats the process with the secondary exception vector. Enabling vectored handlers is discussed in detail in the *OpenVMS Calling Standard*. Because the exception vectors are allocated in static storage, they are not generally used by modular routines.

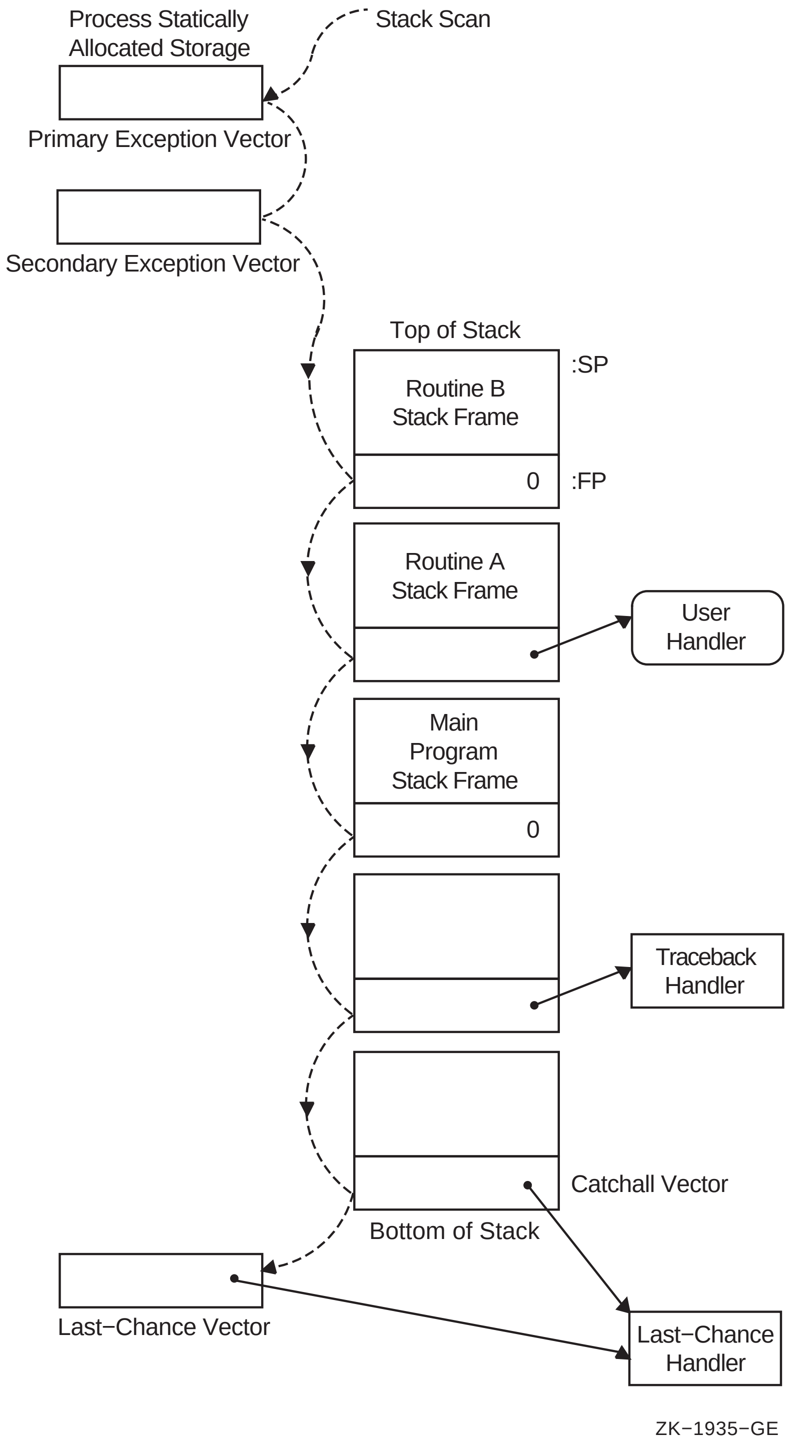
If neither the primary nor secondary vectored handlers handle the exception condition by continuing program execution, then the CHF looks for stack frame condition handlers. It looks for the address of a condition handler in the first longword of the routine stack frame on VAX systems, or in the procedure descriptor (in which the handler valid bit is set) for the routine stack frame on Alpha systems where the exception condition occurred. At this point, several actions are possible, depending on the results of this search:

- If this routine has not set up a condition handler, the CHF continues the stack scan by moving to the previous stack frame (that is, the stack frame of the calling routine).
- If a condition handler is present, the CHF then calls this handler, which may resignal, continue, or unwind. See Section 13.10.

The OpenVMS Condition Handling facility searches for and calls condition handlers from each frame on the stack until the frame pointer is zero (indicating the end of the call sequence). At that point, the CHF calls the vectored catchall handler, which displays an error message and causes the program to exit. Note that, normally, the frame containing the stack catchall handler is at the end of the calling sequence or at the bottom of the stack. Section 13.9 explains the possible actions of default and user condition handlers in more detail.

Figure 13–5 illustrates a stack scan for condition handlers in which the main program calls procedure A, which then calls procedure B. A stack scan is initiated when a hardware exception condition occurs or when a call is made to LIB$SIGNAL or LIB$STOP.

Figure 13–5 Sample Stack Scan for Condition Handlers



13.8.1 Generating Signals with LIB$SIGNAL and LIB$STOP

When software detects an exception condition, the software normally calls one of the run-time library signal-generating routines, LIB$SIGNAL or LIB$STOP, to initiate the signaling mechanism. This call indicates to the calling program that the exception condition has occurred. Your program can also call one of these routines explicitly to indicate an exception condition.

You can signal a condition code by invoking the run-time library procedure LIB$SIGNAL and passing the condition code as the first argument. (The *OpenVMS RTL Library (LIB$) 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 can signal the appropriate condition code rather than return the condition code to the invoking program unit. In addition, some statements also signal condition codes; for example, an assignment statement that attempts to divide by zero signals the condition code SS$_INTDIV.

When your program wants to issue a message and allow execution to continue after handling the condition, it calls the standard routine, LIB$SIGNAL. The calling sequence for LIB$SIGNAL is the following:

```
LIB$SIGNAL condition-value1 [,number1] [,FAO-arg1...,FAO-argn1]
[,condition-value2] [,number2] [,FAO-arg2...,FAO-argn2]
```

Only the **condition-value1** argument must be specified; other arguments are optional. The **number1** argument, if specified, contains the number of FAO (formatted ASCII output) arguments that are associated with **condition-value1**. The **condition-value2** argument may be specified with or without the **number2** or **FAO-arg2** argument. The **number2** argument, if specified, contains the number of FAO arguments that are associated with **condition-value2**. You may specify **condition-value3**, **condition-value4**, **condition-value5**, and so on, along with their corresponding **number** and **FAO** arguments.

condition-value

Operating system usage:	cond_value
type:	longword (unsigned)
access:	read only
mechanism:	by value

A VAX 32-bit condition value. The **condition-value** argument is an unsigned longword that contains this condition value. Section 13.5 explains the format of a VAX condition value.

number1

Operating system usage:	longword_signed
type:	longword integer (signed)
access:	read only
mechanism:	by value

The number of FAO arguments associated with the condition value. The **number** argument is a signed longword integer that contains this number. If you omit the **number** argument or specify it as zero, no FAO arguments follow.

The maximum number of FAO arguments specified must not exceed 253. See Section 13.11 and Section 13.11.3 for more information about FAO arguments.

FAO-arg

Operating system usage:	varying_arg
type:	unspecified
access:	read only
mechanism:	by value

Additional FAO (formatted ASCII output) arguments that are associated with the specified condition value. The **FAO-arg** argument is the address of either a signed longword integer or a character string that contains these additional FAO arguments. Section 13.11 explains the message format.

When your program wants to issue a message and stop execution unconditionally, it calls LIB$STOP. The calling sequence for LIB$STOP is as follows:

```
LIB$STOP condition-value1 [,number1] [,FAO-arg1...,FAO-argn1]
[,condition-value2] [,number2] [,FAO-arg2...,FAO-argn2]
```

Only the **condition-value1** argument must be specified; other arguments are optional. The **number1** argument, if specified, contains the number of FAO arguments that are associated with **condition-value1**. The **condition-value2** argument can be specified with or without the **number2** or **FAO-arg2** argument. The **number2** argument, if specified, contains the number of FAO arguments that are associated with **condition-value2**. You can specify **condition-value3**, **condition-value4**, **condition-value5**, and so on, along with their corresponding **number** and **FAO-arg** arguments.

In both cases, **condition-value** indicates the condition that is being signaled. However, LIB$STOP always sets the severity of **condition-value** to severe before proceeding with the stack-scanning operation.

The FAO arguments describes the details of the exception condition. These are the same arguments that are passed to the OpenVMS Condition Handling facility as part of the signal argument vector. The system default condition handlers pass them to SYS$PUTMSG, which uses them to issue a system message.

Unlike most routines, LIB$SIGNAL and LIB$STOP preserve R0 and R1 as well as the other registers. Therefore, a call to LIB$SIGNAL allows the debugger to display the entire state of the process at the time of the exception condition. This is useful for debugging checks and gathering statistics.

The behavior of LIB$SIGNAL is the same as that of the exception dispatcher that performs the stack scan after hardware detects an exception condition. That is, the system scans the stack in the same way, and the same arguments are passed to each condition handler. This allows a user to write a single condition handler to detect both hardware and software conditions.

13.8.2 Signal Argument Vector

Signaling a condition value causes both VAX and Alpha systems to pass control to a special subprogram called a condition handler. The 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.

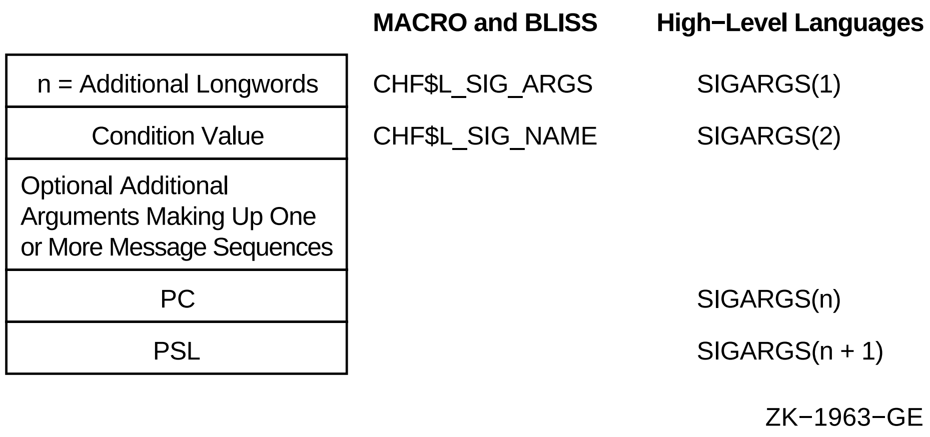
The signal argument vector contains information describing the nature of the hardware or software condition. Figure 13–6 illustrates the open-ended structure of the signal argument vector, which can be from 3 to 257 longwords in length.

The format of the signal argument array and the data it returns is the same on VAX systems and Alpha systems, with the exception of the processor status (PS) returned on Alpha systems and the processor status longword (PSL) returned on VAX systems. On Alpha systems, it is the low-order 32 bits of the PS.

Alpha

On Alpha systems, CHF$IS_SIG_ARGS and CHF$IS_SIG_NAME are aliases for CHF$L_SIG_ARGS and CHF$L_SIG_NAME, as shown in Figure 13–6, and the PSL field for VAX systems is the processor status (PS) field for Alpha systems. ♦

Figure 13–6 Format of the Signal Argument Vector



Fields of the Signal Argument Vector

SIGARGS(1)

An unsigned integer *(n)* designating the number of longwords that follow in the vector, not counting the first, including PC and PSL. (On Alpha systems, the value used for the PSL is the low-order half of the Alpha processor status [PS] register.) For example, the first entry of a 4-longword vector would contain a 3.

SIGARGS(2)

On both VAX systems and Alpha systems, this argument is a 32-bit value that uniquely identifies a hardware or software exception condition. The format of the condition code, which is the same for both VAX systems and Alpha systems, is shown and described in Figure 13–3. However, Alpha systems do not support every condition returned on VAX systems, and Alpha systems define several new conditions that cannot be returned on VAX systems. Table 13–2 lists VAX system condition codes that cannot be returned on Alpha systems.

If more than one message is associated with the error, this is the condition value of the first message. Handlers should always check whether the condition is the one that they expect by examining the STS$V_COND_ID field of the condition value (bits <27:3>). Bits <2:0> are the severity field. Bits <31:28> are control bits; they may have been changed by an intervening handler and so should not be included in the comparison. You can use the RTL routine LIB$MATCH_COND to match the correct fields. If the condition is not expected, the handler should resignal by returning false (bit <0> = 0). The possible exception conditions and their symbolic definitions are listed in Table 13–1.

SIGARGS(3 to n –1)
Optional arguments that provide additional information about the condition. These arguments consist of one or more message sequences. 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 *OpenVMS System Services Reference Manual*. The format of a message sequence is described in Section 13.11.

SIGARGS(n)
The program counter (PC) of the next instruction to be executed if any handler (including the system-supplied handlers) returns with the status SS$_CONTINUE. For hardware faults, the PC is that of the instruction that caused the fault. For hardware traps, the PC is that of the instruction following the one that caused the trap. The error generated by LIB$SIGNAL is a trap. For conditions signaled by calling LIB$SIGNAL or LIB$STOP, the PC is the location following the CALLS or CALLG instruction. See the *VAX Architecture Reference Manual* or *Alpha Architecture Reference Manual* for a detailed description of faults and traps.

SIGARGS(n+1)
On VAX systems the processor status longword (PSL), or on Alpha systems the processor status (PS) register, of the program at the time that the condition was signaled.

For information about the PSL on VAX systems, and the PS on Alpha systems, see the *VAX Architecture Reference Manual* and the *Alpha Architecture Reference Manual*.

Note

LIB$SIGNAL and LIB$STOP copy the variable-length argument list passed by the caller. Then, before calling a condition handler, they append the PC and PSL, or on Alpha systems the processor status (PS) register, entries to the end of the list.

The formats for some conditions signaled by the operating system and the run-time library are shown in Figure 13–7 and Figure 13–8. These formats are the same on VAX systems and Alpha systems, except for the PSL, or on Alpha systems, the PS register.

Figure 13–7 Signal Argument Vector for the Reserved Operand Error Conditions

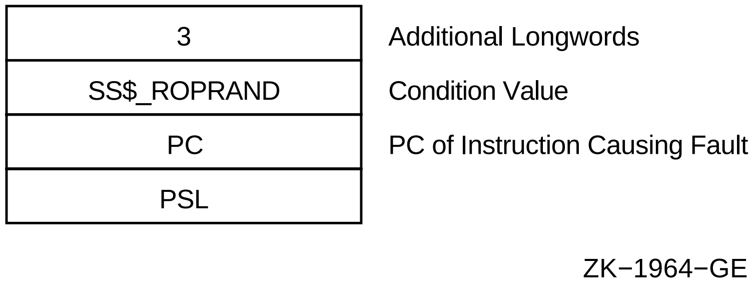
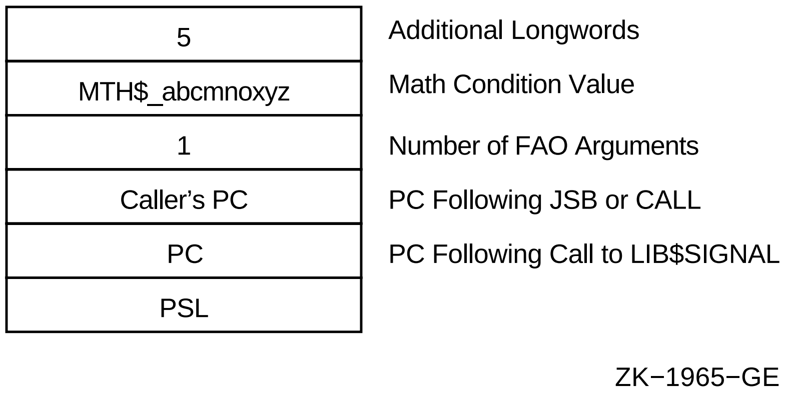


Figure 13–8 Signal Argument Vector for RTL Mathematics Routine Errors



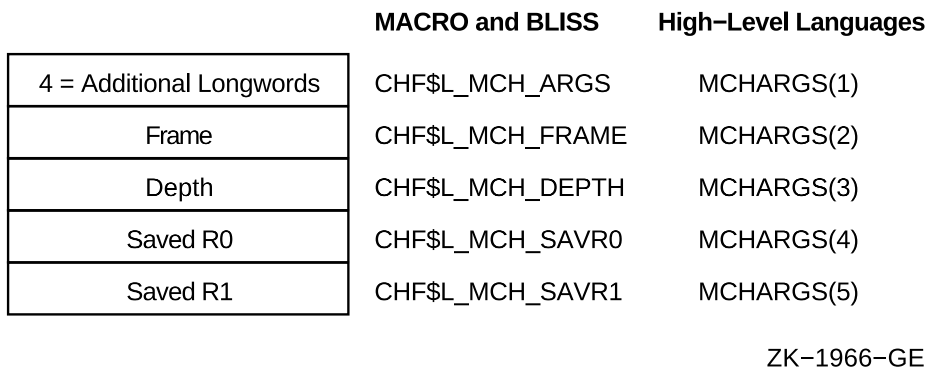
The caller's PC is the PC following the calling program's JSB or CALL to the mathematics routine that detected the error. The PC is that following the call to LIB$SIGNAL.

13.8.3 VAX Mechanism Argument Vector (VAX Only)

VAX

On VAX systems, the mechanism argument vector is a 5-longword vector that contains all of the information describing the state of the process at the time of the hardware or software signaled condition. Figure 13–9 illustrates a mechanism argument vector for VAX systems.

Figure 13–9 Format of a VAX Mechanism Argument Vector



Fields of the VAX Mechanism Argument Vector

MCHARGS(1)
An unsigned integer indicating the number of longwords that follow, not counting the first, in the vector. Currently, this number is always 4.

MCHARGS(2)
The address of the stack frame of the routine that established the handler being called. This address can be used as a base from which to reference the local stack-allocated storage of the establisher, as long as the restrictions on the handler's use of storage are observed. For example, if the call stack is as shown in Figure 13–4, this argument points to the call frame for procedure A.

This value can be used to display local variables in the procedure that established the condition handler if the variables are at known offsets from the frame pointer (FP) of the procedure.

MCHARGS(3)
The stack depth, which is the number of stack frames between the establisher of the condition handler and the frame in which the condition was signaled. To ensure that calls to LIB$SIGNAL and LIB$STOP appear as similar as possible to hardware exception conditions, the call to LIB$SIGNAL or LIB$STOP is not included in the depth.

If the routine that contained the hardware exception condition or that called LIB$SIGNAL or LIB$STOP also handled the exception condition, then the depth is zero; if the exception condition occurred in a called routine and its caller handled the exception condition, then the depth is 1. If a system service signals an exception condition, a handler established by the immediate caller is also entered with a depth of 1.

The following table shows the stack depths for the establishers of condition handlers:

Depth	Meaning
–3	Condition handler was established in the last-chance exception vector.
–2	Condition handler was established in the primary exception vector.
–1	Condition handler was established in the secondary exception vector.

Depth	Meaning
0	Condition handler was established by the frame that was active when the exception occurred.
1	Condition handler was established by the caller of the frame that was active when the exception occurred.
2	Condition handler was established by the caller of the caller of the frame that was active when the exception occurred.
.	
.	
.	

For example, if the call stack is as shown in Figure 13–4, the depth argument passed to handler a would have a value of 2.

The condition handler can use this argument to determine whether to handle the condition. For example, the handler might not want to handle the condition if the exception that caused the condition did not occur in the establisher frame.

MCHARGS(4) and MCHARGS(5)
Copies of the contents of registers R0 and R1 at the time of the exception condition or the call to LIB$SIGNAL or LIB$STOP. When execution continues or a stack unwind occurs, these values are restored to R0 and R1. Thus, a handler can modify these values to change the function value returned to a caller. ♦

13.8.4 Alpha Mechanism Argument Vector (Alpha Only)

Alpha

On Alpha systems, the mechanism array returns much the same data as it does on VAX systems, though its format is changed. The mechanism array returned on Alpha systems preserves the contents of a larger set of integer scratch registers as well as the Alpha floating-point scratch registers. In addition, because Alpha registers are 64 bits long, the mechanism array is constructed of quadwords (64 bits), not longwords (32 bits) as it is on VAX systems. Figure 13–10 shows the format of the mechanism array on Alpha systems.

Figure 13–10 Mechanism Array on Alpha Systems

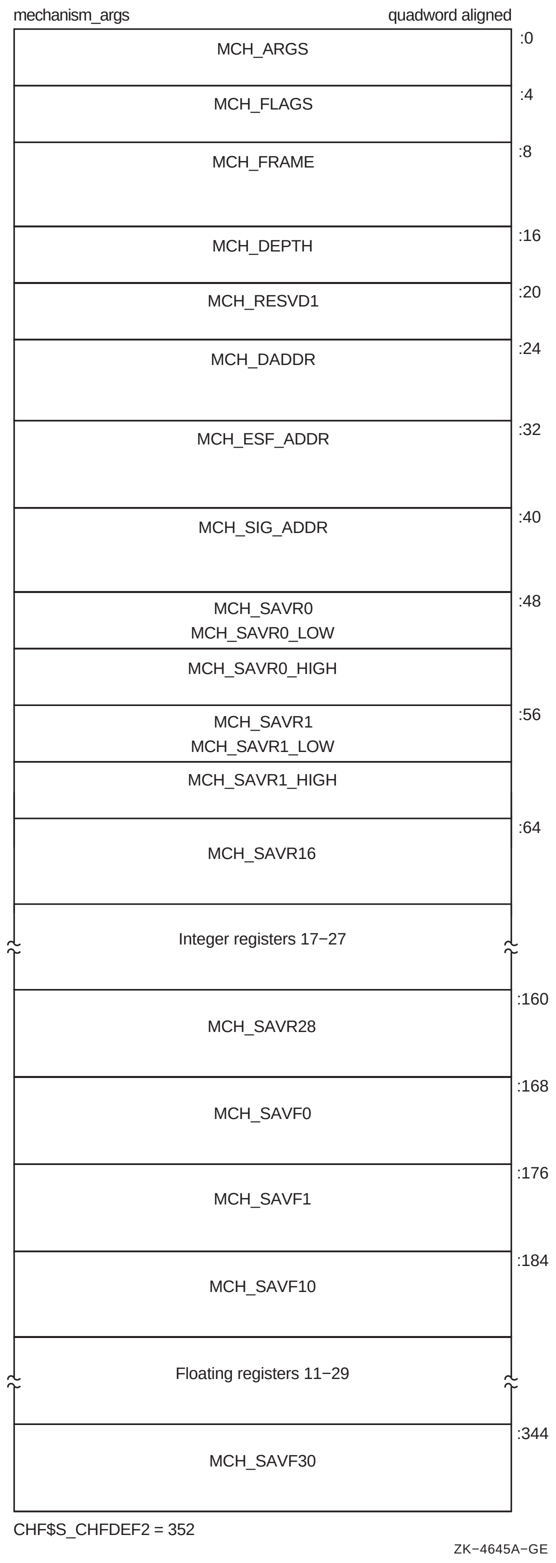


Table 13–11 describes the arguments in the mechanism array.

Table 13–11 Fields in the Alpha Mechanism Array

Argument	Description
CHF$IS_MCH_ARGS	Represents the number of quadwords in the mechanism array, not counting the argument count quadword. (The value contained in this argument is always 43.)
CHF$IS_MCH_FLAGS	Flag bits <63:0> for related argument mechanism information defined as follows for CHF$V_FPREGS:
	Bit <0>: When set, the process has already performed a floating-point operation and the floating-point registers stored in this structure are valid.
	If this bit is clear, the process has not yet performed any floating-point operations, and the values in the floating-point register slots in this structure are unpredictable.
CHF$PH_MCH_FRAME	Contains the frame pointer (FP) in the procedure context of the establisher.
CHF$IS_MCH_DEPTH	Positive count of the number of procedure activation stack frames between the frame in which the exception occurred and the frame depth that established the handler being called.
CHF$PS_MCH_DADDR	Address of the handler data quadword if the exception handler data field is present (as indicated by PDSC.FLAGS.HANDLER_DATA_VALID); otherwise, contains zero.
CHF$PH_MCH_ESF_ADDR	Address of the exception stack frame (see the *Alpha Architecture Reference Manual*).
CHF$PH_MCH_SIG_ADDR	Address of the signal array. The signal array is a 32-bit (longword) array.
CHF$IH_MCH_SAVR*nn*	Contains a copy of the saved integer registers at the time of the exception. The following registers are saved: R0, R1, and R16–R28. Registers R2–R15 are implicitly saved in the call chain.
CHF$FM_MCH_SAVF*nn*	Contains a copy of the saved floating-point registers at the time of the exception or may have unpredictable data as described in CHF$IS_MCH_FLAGS. If the floating-point register fields are valid, the following registers are saved: F0, F1, and F10–F30. Registers F2–F9 are implicitly saved in the call chain.

For more information and recommendations about using the mechanism argument vector on Alpha systems, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*. ♦

13.8.5 Multiple Active Signals

A signal is said to be active until the routine that signaled regains control or until the stack is unwound or the image exits. A second signal can occur while a condition handler or a routine it has called is executing. This situation is called **multiple active signals** or **multiple exception conditions**. When this situation occurs, the stack scan is not performed in the usual way. Instead, the frames that were searched while processing all of the previous exception conditions are skipped when the current exception condition is processed. This is done in order to avoid recursively reentering a routine that is not reentrant. For example, Fortran code typically is not recursively reentrant. If a Fortran handler were called while another activation of that handler was still going, the results would be unpredictable.

A second exception may occur while a condition handler or a procedure that it has called is still executing. In this case, when the exception dispatcher searches for a condition handler, it skips the frames that were searched to locate the first handler.

The search for a second handler terminates in the same manner as the initial search, as described in Section 13.6.

If the SYS$UNWIND system service is issued by the second active condition handler, the depth of the unwind is determined according to the same rules followed in the exception dispatcher's search of the stack: all frames that were searched for the first condition handler are skipped.

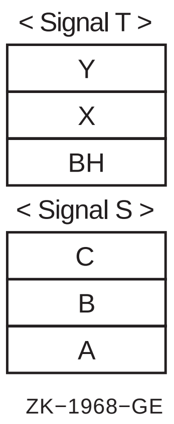
Primary and secondary vectored handlers, on the other hand, are always entered when an exception occurs.

If an exception occurs during the execution of a handler that was established in the primary or secondary exception vector, that handler must handle the additional condition. Failure to do so correctly might result in a recursive exception loop in which the vectored handler is repeatedly called until the user stack is exhausted.

The modified search routine is best illustrated with an example. Assume the following calling sequence:

1. Routine A calls routine B, which calls routine C.
2. Routine C signals an exception condition (signal S), and the handler for routine C (CH) resignals.
3. Control passes to BH, the handler for routine B. The call frame for handler BH is located on top of the signal and mechanism arrays for signal S. The saved frame pointer in the call frame for BH points to the frame for routine C.
4. BH calls routine X; routine X calls routine Y.
5. Routine Y signals a second exception condition (signal T). Figure 13–11 illustrates the stack contents after the second exception condition is signaled.

Figure 13–11 Stack After Second Exception Condition Is Signaled



Normally, the OpenVMS Condition Handling facility (CHF) searches all currently active frames for condition handlers, including B and C. If this happens, however, BH is called again. At this point, you skip the condition handlers that have already been called. Thus, the search for condition handlers should proceed in the following order:

YH
XH
BHH (the handler for routine B's handler)
AH

6. The search now continues in its usual fashion. The CHF examines the primary and secondary exception vectors, then frames Y, X, and BH. Thus, handlers YH, XH, and BHH are called. Assume that these handlers resignal.
7. The CHF now skips the frames that have already been searched and resumes the search for condition handlers in routine A's frame. The depths that are passed to handlers as a result of this modified search are 0 for YH, 1 for XH, 2 for BHH, and 3 for AH.

Because of the possibility of multiple active signals, you should be careful if you use an exception vector to establish a condition handler. Vectored handlers are called, not skipped, each time an exception occurs.

13.9 Types of Condition Handlers

VAX

On VAX systems, when a routine is activated, the first longword in its stack frame is set to 0. This longword is reserved to contain an address pointing to another routine called the condition handler. If an exception condition is signaled during the execution of the routine, the OpenVMS Condition Handling Facility uses the address in the first longword of the frame to call the associated condition handler. ♦

Alpha

On Alpha systems, each procedure, other than a null frame procedure, can have a condition handler potentially associated with it, identified by the HANDLER_VALID, STACK_HANDLER, or REG_HANDLER field of the associated procedure descriptor. You establish a handler by including the procedure value of the handler procedure in that field. ♦

The arguments passed to the condition-handling routine are the signal and mechanism argument vectors, described in Section 13.8.2, Section 13.8.3, and Section 13.8.4.

Various types of condition handlers can be called for a given routine:

- User-supplied condition handlers

  You can write your own condition handler and set up its address in the stack frame of your routine using the run-time library routine LIB$ESTABLISH or the mechanism supplied by your language.

  Alpha On Alpha systems, LIB$ESTABLISH is not supported, though high-level languages may support it for compatibility. ♦

- Language-supplied condition handlers

  Many high-level languages provide a means for setting up handlers that are global to a single routine. If your language provides a condition-handling mechanism, you should always use it. If you also try to establish a condition handler using LIB$ESTABLISH, the two methods of handling exception conditions conflict, and the results are unpredictable.

- System default condition handlers

  The operating system provides a set of default condition handlers. These take over if there are no other condition handler addresses on the stack, or if all the previous condition handlers have passed on (resignaled) the indication of the exception condition.

13.9.1 Default Condition Handlers

The operating system establishes the following default condition handlers each time a new image is started. The default handlers are shown in the order they are encountered when the operating system processes a signal. These three handlers are the only handlers that output error messages.

- Traceback handler

  The traceback handler is established on the stack after the catchall handler. This enables the traceback handler to get control first. This handler performs three functions in the following order:

  1. Outputs an error message using the Put Message (SYS$PUTMSG) system service. SYS$PUTMSG formats the message using the Formatted ASCII Output (SYS$FAO) system service and sends the message to the devices SYS$ERROR and SYS$OUTPUT (if it differs from SYS$ERROR). That is, it 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.)
  2. Issues a symbolic traceback, which shows the state of the routine stack at the time of the exception condition. That is, it displays the names of the program units in the calling hierarchy and the line numbers of the invocation statements.

3. Decides whether to continue executing the image or to force an exit based on the severity field of the condition value:

Severity	Error Type	Action
1	Success	Continue
3	Information	Continue
0	Warning	Continue
2	Error	Continue
4	Severe	Exit

   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.

- Catchall handler

  The operating system establishes the catchall handler in the first stack frame and thus calls it last. This handler performs the same functions as the traceback handler except for the stack traceback. That is, it issues an error message and decides whether to continue execution. The catchall is called only if you link with the /NOTRACEBACK qualifier. It displays the message associated with the condition code and then continues program execution or, if the error is severe, terminates execution.

- Last-chance handler

  The operating system establishes the last-chance handler with a system exception vector. In most cases, this vector contains the address of the catchall handler, so that these two handlers are actually the same. The last-chance handler is called only if the stack is invalid or all the handlers on the stack have resignaled. If the debugger is present, the debugger's own last-chance handler replaces the system last-chance 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.

In the following 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.

13.9.2 Interaction Between Default and User-Supplied Handlers

Several results are possible after a routine signals, depending on a number of factors, such as the severity of the error, the method of generating the signal, and the action of the condition handlers you have defined and the default handlers. Given the severity of the condition and the method of signaling, Figure 13–12 lists all combinations of interaction between user condition handlers and default condition handlers.

Figure 13–12 Interaction Between Handlers and Default Handlers

Severity of Condition	**User Handler Specifies CONTINUE**	**User Handler Specifies UNWIND**	**Default Handler Gets Control**	**No Handler Found (Bad Stack)**
Exception Condition Is Signaled by a Call to LIB$SIGNAL or Detected by Hardware				
WARNING, INFO, or ERROR	RETURN	UNWIND	Issue Condition Message RETURN	Call Last−Chance Handler EXIT
SEVERE	RETURN	UNWIND	Issue Condition Message EXIT	Call Last−Chance Handler EXIT
Exception Condition Is Signaled by a Call to LIB$STOP				
LIB$STOP Forces Severity to SEVERE	Message: "Attempt to continue from stop" EXIT	UNWIND	Issue Condition Message EXIT	Call Last−Chance Handler EXIT

ZK−4257−GE

13.10 Types of Actions Performed by Condition Handlers

When a condition handler returns control to the OpenVMS Condition Handling facility (CHF), the facility takes one of the following types of actions, depending on the value returned by the condition handler:

- Signal a condition

  Signaling a condition initiates the search for an established condition handler.
- Continue

  The condition handler may or may not be able to fix the problem, but the program can attempt to continue execution. The handler places the return status value SS$_CONTINUE in R0 and issues a RET instruction to return control to the dispatcher. If the exception was a fault, the instruction that caused it is reexecuted; if the exception was a trap, control is returned at the instruction following the one that caused it. A condition handler cannot continue if the exception condition was signaled by calling LIB$STOP.

  Section 13.12.1 contains more information about continuing.
- Resignal

  The handler cannot fix the problem, or this condition is one that it does not handle. It places the return status value SS$_RESIGNAL in R0 and issues a RET instruction to return control to the exception dispatcher. The dispatcher resumes its search for a condition handler. If it finds another condition handler, it passes control to that routine. A handler can alter the severity of the signal before resignaling.

  Section 13.12.2 contains more information about resignaling
- Unwind

  The condition handler cannot fix the problem, and execution cannot continue while using the current flow. The handler issues the Unwind Call Stack (SYS$UNWIND) system service to unwind the call stack. Call frames can then be removed from the stack and the flow of execution modified, depending on the arguments to the SYS$UNWIND service.

  When a condition handler has already called SYS$UNWIND, any return status from the condition handler is ignored by the CHF. The CHF now unwinds the stack.

  Unwinding the routine call stack removes call frames, starting with the frame in which the condition occurred, and returns control to an earlier routine in the calling sequence. You can unwind the stack whether the condition was detected by hardware or signaled using LIB$SIGNAL or LIB$STOP. Unwinding is the only way to continue execution after a call to LIB$STOP.

  Section 13.12.3 describes how to write a condition handler that unwinds the call stack.

Alpha

- Perform a nonlocal GOTO unwind

  On Alpha systems, a GOTO unwind operation is a transfer of control that leaves one procedure invocation and continues execution in a prior (currently active) procedure. This unified GOTO operation gives unterminated procedure invocations the opportunity to clean up in an orderly way. See Section 13.10.2 for more information about GOTO unwind operations. ♦

13.10.1 Unwinding the Call Stack

One type of action a condition handler can take is to unwind the procedure call stack. The unwind operation is complex and should be used only when control must be restored to an earlier procedure in the calling sequence. Moreover, use of the SYS$UNWIND system service requires the calling condition handler to be aware of the calling sequence and of the exact point to which control is to return.

SYS$UNWIND accepts two optional arguments:

- The depth to which the unwind is to occur. If the depth is 1, the call stack is unwound to the caller of the procedure that incurred the exception. If the depth is 2, the call stack is unwound to the caller's caller, and so on. By specifying the depth in the mechanism array, the handler can unwind to the procedure that established the handler.
- The address of a location to receive control when the unwind operation is complete, that is, a PC to replace the current PC in the call frame of the procedure that will receive control when all specified frames have been removed from the stack.

If no argument is supplied to SYS$UNWIND, the unwind is performed to the caller of the procedure that established the condition handler that is issuing the SYS$UNWIND service. Control is returned to the address specified in the return PC for that procedure. Note that this is the default and the normal case for unwinding.

Another common case of unwinding is to unwind to the procedure that declared the handler. On VAX systems, this is done by using the depth value from the exception mechanism array (CHF$L_MCH_DEPTH) as the depth argument to SYS$UNWIND. On Alpha systems, this is done by using the depth value from the exception mechanism array (CHF$IS_MCH_DEPTH) as the depth argument to SYS$UNWIND.

Therefore, it follows that the default unwind (no depth specified) is equivalent to specifying CHF$L_MCH_DEPTH plus 1 on VAX systems. On Alpha systems, the default unwind (no depth specified) is equivalent to specifying CHF$IS_MCH_DEPTH plus 1. In certain instances of nested exceptions, however, this is not the case. Digital recommends that you omit the depth argument when unwinding to the caller of the routine that established the condition handler.

Figure 13–13 illustrates an unwind situation and describes some of the possible results.

The unwind operation consists of two parts:

1. In the call to SYS$UNWIND, the return PCs saved in the stack are modified to point into a routine within the SYS$UNWIND service, but the entire stack remains present.
2. When the handler returns, control is directed to this routine by the modified PCs. It proceeds to return to itself, removing the modified stack frames, until the stack has been unwound to the proper depth.

For this reason, the stack is in an intermediate state directly after calling SYS$UNWIND. Handlers should, in general, return immediately after calling SYS$UNWIND.

During the actual unwinding of the call stack, the unwind routine examines each frame in the call stack to see whether a condition handler has been declared. If a handler has been declared, the unwind routine calls the handler with the status value SS$_UNWIND (indicating that the call stack is being unwound) in the condition name argument of the signal array. When a condition handler is called with this status value, it can perform any procedure-specific cleanup operations required. For example, the handler should deallocate any processwide resources that have been allocated. Then, the handler returns control to the OpenVMS Condition Handling facility. After the handler returns, the call frame is removed from the stack.

When a condition handler is called during the unwinding operation, the condition handler must not generate a new signal. A new signal would result in unpredictable behavior.

Thus, in Figure 13–13, handler B can be called a second time, during the unwind operation. Note that handler B does not have to be able to interpret the SS$_ UNWIND status value specifically; the RET instruction merely returns control to the unwind procedure, which does not check any status values.

Handlers established by the primary, secondary, or last-chance vector are not called, because they are not removed during an unwind operation.

While it is unwinding the stack, the OpenVMS Condition Handling facility ignores any function value returned by a condition handler. For this reason, a handler cannot both resignal and unwind. Thus, the only way for a handler to both issue a message and perform an unwind is to call LIB$SIGNAL and then call $UNWIND. If your program calls $UNWIND before calling LIB$SIGNAL, the result is unpredictable.

When the OpenVMS Condition Handling facility calls the condition handler that was established for each frame during unwind, the call is of the standard form, described in Section 13.2. The arguments passed to the condition handler (the signal and mechanism argument vectors) are shown in Section 13.8.2, Section 13.8.3, and Section 13.8.4.

VAX

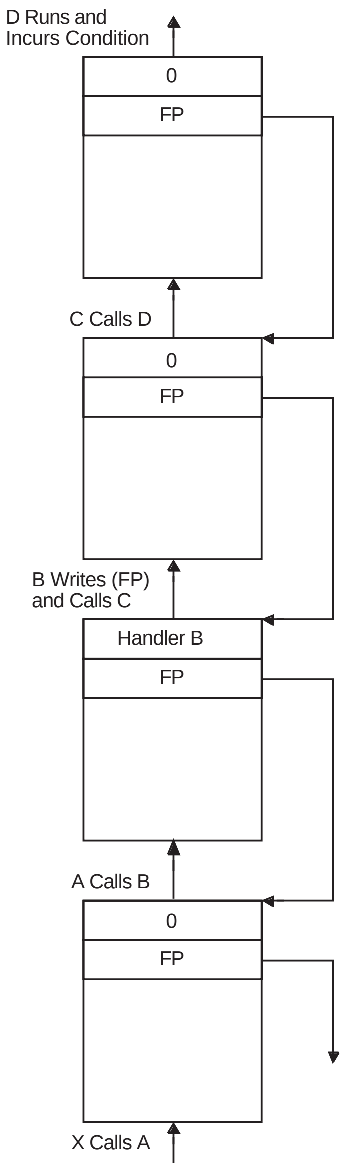
On VAX systems, if the handler is to specify the function value of the last function to be unwound, it should modify the saved copies of R0 and R1 (CHF$L_MCH_ SAVR0 and CHF$L_MCH_SAVR1) in the mechanism argument vector. ♦

Alpha

On Alpha systems, the handler should modify the saved copies of R0 and R1 (CHF$IH_MCH_SAVR*nn*). ♦

R0 and R1 are restored from the mechanism argument vector at the end of the unwind.

Figure 13–13 Unwinding the Call Stack



1 The procedure call stack is as shown. Assume that no exception vectors are declared for the process and that the exception occurs during the execution of procedure D.

2 Because neither procedure D nor procedure C has established a condition handler, handler B receives control.

3 If handler B issues the $UNWIND system service with no arguments, the call frames for B, C, and D are removed from the stack (along with the call frame for handler B itself), and control returns to procedure A. Procedure A receives control at the point following its call to procedure B.

4 If handler B issues the $UNWIND system service specitying a depth of 2, call frames for C and D are removed, and control returns to procedure B.

ZK−0860−GE

13.10.2 GOTO Unwind Operations (Alpha Only)

Alpha

A current procedure invocation is one in whose context the thread of execution is currently executing. At any instant, a thread of execution has exactly one current procedure. If code in the current procedure calls another procedure, then the called procedure becomes the current procedure. As each stack frame or register frame procedure is called, its invocation context is recorded in a procedure frame. The invocation context is mainly a snapshot of process registers at procedure invocation. It is used during return from the called procedure to restore the calling procedure's state. The chain of all procedure frames starting with the current procedure and going all the way back to the first procedure invocation for the thread is called the **call chain**. While a procedure is part of the call chain, it is called an **active procedure**.

When a current procedure returns to its calling procedure, the most recent procedure frame is removed from the call chain and used to restore the now current procedure's state. As each current procedure returns to its calling procedure, its associated procedure frame is removed from the call chain. This is the normal unwind process of a call chain.

You can bypass the normal return path by forcibly unwinding the call chain. The Unwind Call Chain (SYS$UNWIND) system service allows a condition handler to transfer control from a series of nested procedure invocations to a previous point of execution, bypassing the normal return path. The Goto Unwind (SYS$GOTO_UNWIND) system service allows any procedure to achieve the same effect. SYS$GOTO_UNWIND restores saved register context for each nested procedure invocation, calling the condition handler, if any, for each procedure frame that it unwinds. Restoring saved register context from each procedure frame from the most recent one to the target procedure frame ensures that the register context is correct when the target procedure gains control. Also, each condition handler called during unwind can release any resources acquired by its establishing procedure.

For information about the GOTO unwind operations and how to use the SYS$GOTO_UNWIND system service, see the *OpenVMS Calling Standard* and the *OpenVMS System Services Reference Manual*. ♦

13.11 Displaying Messages

The standard format for a message is as follows:

%facility-l-ident, message-text

facility	Abbreviated name of the software component that issued the message
l	Indicator showing the severity level of the exception condition that caused the message
ident	Symbol of up to nine characters representing the message
message-text	Brief definition of the cause of the message

The message can also include up to 255 formatted ASCII output (FAO) arguments. These arguments can be used to display variable information about the condition that caused the message. In the following examples, the file specification is an FAO argument:

```
%TYPE-W-OPENIN, error opening _DB0:[FOSTER]AUTHOR.DAT; as input
```

For information about specifying FAO parameters, see Section 13.11.4.3.

Signaling

Signaling provides a consistent and unified method for displaying messages. This section describes how the OpenVMS Condition Handling facility translates the original signal into intelligible messages.

Signaling is used to signal exception conditions generated by Digital software. When software detects an exception condition, it signals the exception condition to the user by calling LIB$SIGNAL or LIB$STOP. The signaling routine passes a signal argument list to these run-time library routines. This signal argument list is made up of the condition value and a set of optional arguments that provide information to condition handlers.

You can use the signaling mechanism to signal messages that are specific to your application. Further, you can chain your own message to a system message. For more information, see Section 13.11.3.

LIB$SIGNAL and LIB$STOP copy the signal argument list and use it to create the signal argument vector. The signal argument vector serves as part of the input to the user-established handlers and the system default handlers.

If all intervening handlers have resignaled, the system default handlers take control. The system-supplied default handlers are the only handlers that should actually issue messages, whether the exception conditions are signaled by Digital software or your own programs. That is, a routine should signal exception conditions rather than issue its own messages. In this way, other applications can call the routine and override its signal in order to change the messages. Further, this technique decides formatting details, and it also keeps wording centralized and consistent.

The system default handlers pass the signal argument vector to the Put Message (SYS$PUTMSG) system service. SYS$PUTMSG formats and displays the information in the signal argument vector.

SYS$PUTMSG performs the following steps:

1. Interprets the signal argument vector as a series of one or more message sequences. Each message sequence starts with a 32-bit, systemwide condition value that identifies a message in the system message file. SYS$PUTMSG interprets the message sequences according to type defined by the facility of the condition.
2. Obtains the text of the message using the Get Message (SYS$GETMSG) system service. The message text definition is actually a SYS$FAO control string. It may contain embedded FAO directives. These directives determine how the FAO arguments in the signal argument vector are formatted. (For more information about SYS$FAO, see the *OpenVMS System Services Reference Manual*.)
3. Calls SYS$FAO to format the message, substituting the values from the signal argument list.
4. Issues the message on device SYS$OUTPUT. If SYS$ERROR is different from SYS$OUTPUT, and the severity field in the condition value is not success, $PUTMSG also issues the message on device SYS$ERROR.

You can use the signal array that the operating system passes to the condition handler as the first argument of the SYS$PUTMSG system service. The signal array contains the condition code, the number of required FAO arguments for each condition code, and the FAO arguments (see Figure 13–14). The *OpenVMS System Services Reference Manual* contains complete specifications for SYS$PUTMSG.

See Section 13.11.2 for information about how to create and suppress messages on a running log using SYS$PUTMSG.

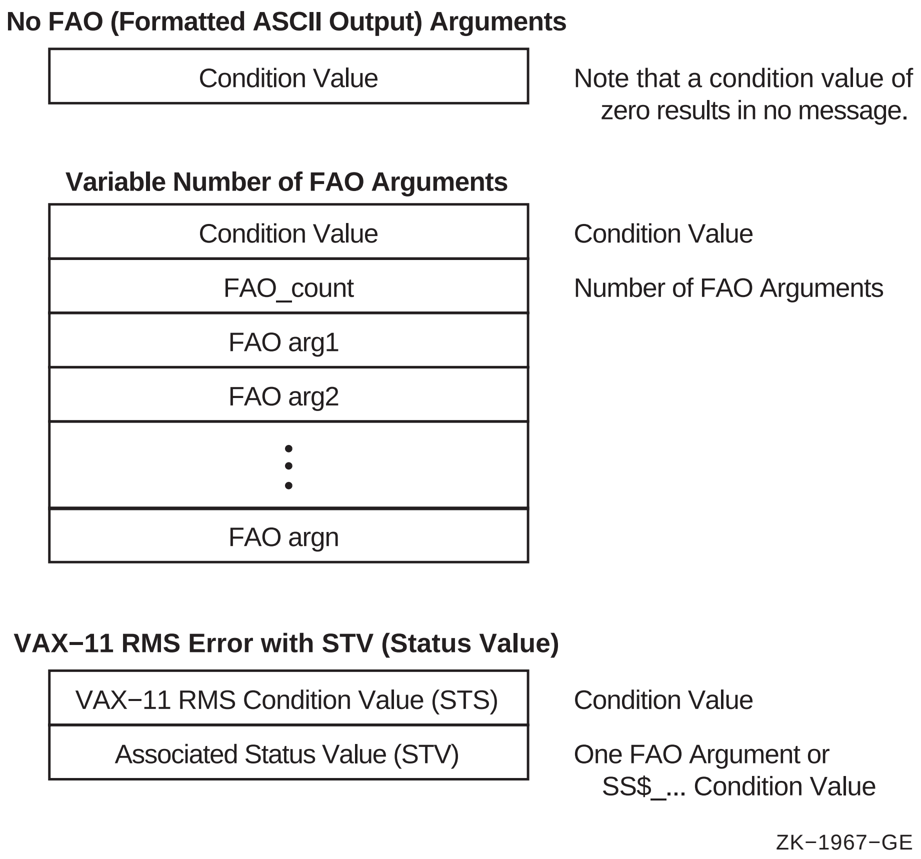
The last two array elements, the PC and PSL, are not FAO arguments and should be deleted before the array is passed to SYS$PUTMSG. Because the first element of the signal array contains the number of longwords in the array, you can effectively delete the last two elements of the array by subtracting 2 from the value in the first element. Before exiting from the condition handler, you should restore the last two elements of the array by adding 2 to the first element in case other handlers reference the array.

In the following example, 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
```

Each message sequence in the signal argument list produces one line of output. Figure 13–14 illustrates the three possible message sequence formats.

Figure 13–14 Formats of Message Sequences


No FAO (Formatted ASCII Output) Arguments
Condition Value
Note that a condition value of zero results in no message.
Variable Number of FAO Arguments
Condition Value
FAO_count
FAO arg1
FAO arg2
FAO argn
Condition Value
Number of FAO Arguments
VAX−11 RMS Error with STV (Status Value)
VAX−11 RMS Condition Value (STS)
Associated Status Value (STV)
Condition Value
One FAO Argument or SS$_... Condition Value
ZK−1967−GE

OpenVMS RMS system services return two related completion values: the completion code and the associated status value. The completion code is returned in R0 using the function value mechanism. The same value is also placed in the Completion Status Code field of the RMS file access block (FAB) or record access block (RAB) associated with the file (FAB$L_STS or RAB$L_STS). The status value is returned in the Status Value field of the same FAB or RAB (FAB$L_STV or RAB$L_STV). The meaning of this secondary value is based on the corresponding STS (Completion Status Code) value. Its meaning could be any of the following:

- An operating system condition value of the form SS$_ . . .
- An RMS value, such as the size of a record that exceeds the buffer size
- Zero

Rather than have each calling program determine the meaning of the STV value, SYS$PUTMSG performs the necessary processing. Therefore, this STV value must always be passed in place of the FAO argument count. In other words, an RMS message sequence always consists of two arguments (passed by immediate value): an STS value and an STV value.

13.11.1 Chaining Messages

You can use a condition handler to add condition values 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 operating system signals the condition code SS$_INTDIV when the program attempts to divide by zero. (In calculating the standard deviation, the divisor is the number of values entered minus 1.) You could use a condition handler to add a user-defined message to the original message to indicate that only one value was entered.

To display multiple messages, pass the condition values associated with the messages to the RTL routine LIB$SIGNAL. To display the message associated with an additional condition code, the handler must pass LIB$SIGNAL the condition code, the number of FAO arguments used, and the FAO arguments. To display the message associated with the originally signaled condition codes, the handler must pass LIB$SIGNAL each element of the signal array as a separate argument. Because the signal array is a variable-length array and LIB$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 values and FAO information from the signal array to your new array. The first element and the last two elements of the signal array do not contain FAO information and should not be transferred.
3. Fill any remaining elements of your new array with zeros.

The following example demonstrates steps 2 and 3:

```
DO I = 1, 15

  IF (I .LE. SIGARGS(1) - 2) THEN
    NEWSIGARGS(I) = SIGARGS(I+1)  ! Start with SIGARGS(2)
    ELSE
    NEWSIGARGS(I) = 0             ! Pad with zeros
  END IF

END DO
```

Because 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
```

A signal argument list may contain one or more condition values and FAO arguments. Each condition value and its FAO arguments is "chained" to the next condition value and its FAO arguments. You can use chained messages to provide more specific information about the exception condition being signaled, along with a general message.

The following message source file defines the exception condition PROG__FAIGETMEM:

```
     .FACILITY     PROG,1 /PREFIX=PROG__

     .SEVERITY     FATAL
     .BASE     100

     FAIGETMEM   <failed to get !UL bytes of memory>/FAO_COUNT=1

     .END
```

This source file sets up the exception message as follows:

- The .FACILITY directive specifies the facility, PROG, and its number, 1. It also adds the /PREFIX qualifier to determine the prefix to be used in the message.

- The .SEVERITY directive specifies that PROG__FAIGETMEM is a fatal exception condition. That is, the SEVERITY field in the condition value for PROG__FAIGETMEM is set to severe (bits <0:3> = 4).
- The BASE directive specifies that the condition identification numbers in the PROG facility will begin with 100.
- FAIGETMEM is the symbol name. This name is combined with the prefix defined in the facility definition to make the message symbol. The message symbol becomes the symbolic name for the condition value.
- The text in angle brackets is the message text. This is actually a SYS$FAO control string. When $PUTMSG calls the $FAO system service to format the message, $FAO includes the FAO argument from the signal argument vector and formats the argument according to the embedded FAO directive (!UL).
- The .END statement terminates the list of messages for the PROG facility.

13.11.2 Logging Error Messages to a File

You can write a condition handler to obtain a copy of a system error message text and write the message into an auxiliary file, such as a listing file. In this way, you can receive identical messages at the terminal (or batch log file) and in the auxiliary file.

To log messages, you must write a condition handler and an action subroutine. Your handler calls the Put Message (SYS$PUTMSG) system service explicitly. The operation of SYS$PUTMSG is described in Section 13.11. The handler passes to SYS$PUTMSG the signal argument vector and the address of the action subroutine. SYS$PUTMSG passes to the action subroutine the address of a string descriptor that contains the length and address of the formatted message. The action subroutine can scan the message or copy it into a log file, or both.

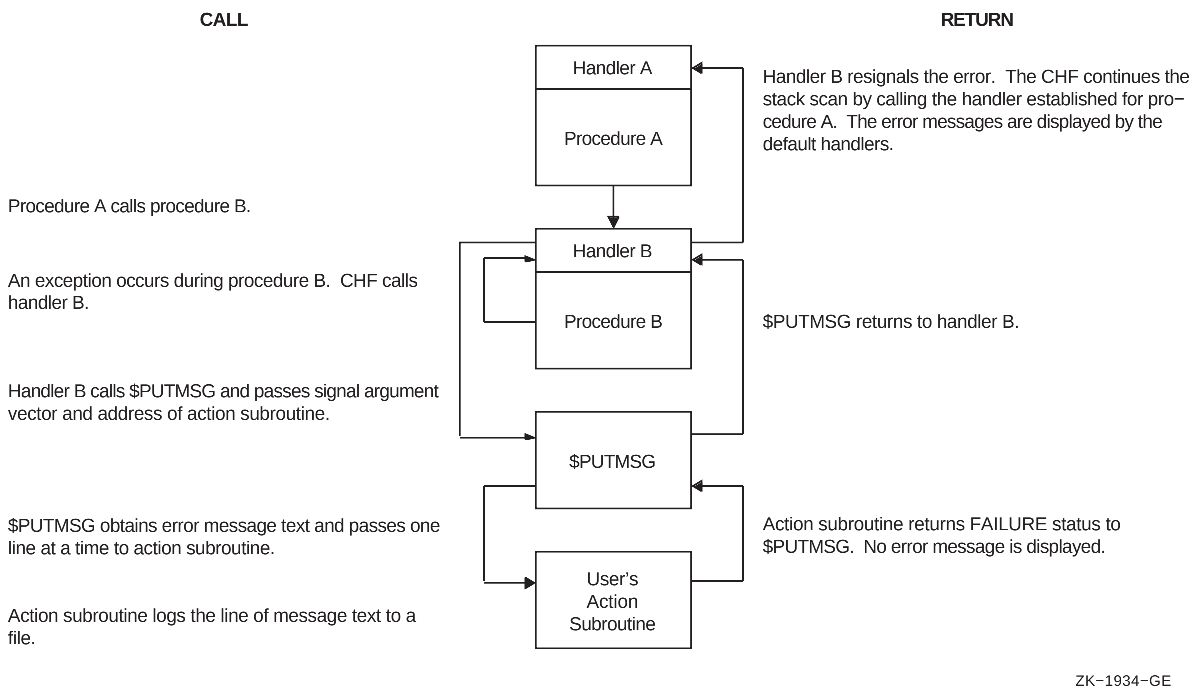
It is important to keep the display messages centralized and consistent. Thus, you should use only SYS$PUTMSG to display or log system error messages. Further, because the system default handlers call SYS$PUTMSG to display error messages, your handlers should avoid displaying the error messages. You can do this in two ways:

- Your handler should not call SYS$PUTMSG directly to display an error message. Instead, your handler should resignal the error. This allows other calling routines to change or suppress the message or to recover from the error. The system default condition handlers display the message.
- If the action subroutine that you supply to SYS$PUTMSG returns a success code, SYS$PUTMSG displays the error message on SYS$OUTPUT or SYS$ERROR, or both. When a program executes interactively or from within a command procedure, the logical names SYS$OUTPUT and SYS$ERROR are both equated to the user's terminal by default. Thus, your action routine should process the message and then return a failure code so that SYS$PUTMSG does not display the message at this point.

  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.

Figure 13–15 shows the sequence of events involved in calling SYS$PUTMSG to log an error message to a file.

Figure 13–15 Using a Condition Handler to Log an Error Message


CALL
RETURN
Handler A
Procedure A
Handler B resignals the error. The CHF continues the stack scan by calling the handler established for pro−cedure A. The error messages are displayed by the default handlers.
Procedure A calls procedure B.
Handler B
Procedure B
An exception occurs during procedure B. CHF calls handler B.
$PUTMSG returns to handler B.
Handler B calls $PUTMSG and passes signal argument vector and address of action subroutine.
$PUTMSG
$PUTMSG obtains error message text and passes one line at a time to action subroutine.
Action subroutine returns FAILURE status to $PUTMSG. No error message is displayed.
User's Action Subroutine
Action subroutine logs the line of message text to a file.
ZK−1934−GE

13.11.2.1 Creating a Running Log of Messages Using SYS$PUTMSG

To keep a running log (that is, a log that is resumed each time your program is invoked) of the messages displayed by your program, use SYS$PUTMSG. Create a condition handler that invokes SYS$PUTMSG regardless of the signaled condition code. When you invoke SYS$PUTMSG, specify a function that writes the formatted message to your log file and then returns with a function value of 0. Have the condition handler resignal the condition code. One of the arguments of SYS$PUTMSG 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 *OpenVMS System Services Reference Manual* contains complete specifications for SYS$PUTMSG.

13.11.2.2 Suppressing the Display of Messages in the Running Log

To keep a running log of messages, you might have your main program open a file for the error log, write the date, and then establish a condition handler to write all signaled messages to the error log. Each time a condition is signaled, a condition handler like the one in the following example invokes SYS$PUTMSG and specifies a function that writes the message to the log file and returns with a function value of 0. SYS$PUTMSG writes the message to the log file but does not display the message. After SYS$PUTMSG writes the message to the log file, the condition handler resignals to continue program execution. (The condition handler uses LIB$GET_COMMON to read the unit number of the file from the per-process common block.)

ERR.FOR

```
INTEGER FUNCTION ERRLOG (SIGARGS,
2                        MECHARGS)
! Writes the message to file opened on the
! logical unit named in the per-process common block
! Define the dummy arguments
INTEGER SIGARGS(*),
2       MECHARGS(*)
INCLUDE '($SSDEF)'

EXTERNAL PUT_LINE
INTEGER PUT_LINE
! Pass signal array and PUT_LINE routine to SYS$PUTMSG
SIGARGS(1) = SIGARGS(1) - 2   ! Subtract PC/PSL from signal array
CALL SYS$PUTMSG (SIGARGS,
2                PUT_LINE, )
SIGARGS(1) = SIGARGS(1) + 2   ! Replace PC/PSL

ERRLOG = SS$_RESIGNAL

END
```

PUT_LINE.FOR

```
INTEGER FUNCTION PUT_LINE (LINE)
! Writes the formatted message in LINE to
! the file opened on the logical unit named
! in the per-process common block
! Dummy argument
CHARACTER*(*) LINE
! Logical unit number
CHARACTER*4 LOGICAL_UNIT
INTEGER UNIT_NUM
! Indicates that SYS$PUTMSG is not to display the message
PUT_LINE = 0
! Get logical unit number and change to integer
STATUS = LIB$GET_COMMON (LOGICAL_UNIT)
READ (UNIT = LOGICAL_UNIT,
2     FMT = '(I4)') UNIT_NUMBER
! The main program opens the error log
WRITE (UNIT = UNIT_NUMBER,
2      FMT = '(A)') LINE

END
```

13.11.3 Using the Message Utility to Signal and Display User-Defined Messages

Section 13.11 explains how the OpenVMS Condition Handling facility displays messages. The signal argument list passed to LIB$SIGNAL or LIB$STOP can be seen as one or more message sequences. Each message sequence consists of a condition value; an FAO count, which specifies the number of FAO arguments to come; and the FAO arguments themselves. (The FAO count is omitted in the case of system and RMS messages.) The message text definition itself is actually a SYS$FAO control string, which may contain embedded $FAO directives. The *OpenVMS System Services Reference Manual* describes the Formatted ASCII Output (SYS$FAO) system service in detail.

The Message utility is provided for compiling message sequences specific to your application. When you have defined an exception condition and used the Message utility to associate a message with that exception condition, your program can call LIB$SIGNAL or LIB$STOP to signal the exception condition. You signal a message that is defined in a message source file by calling LIB$SIGNAL or LIB$STOP, as for any software-detected exception condition. Then the system default condition handlers display your error message in the standard operating system format.

To use the Message utility, follow these steps:

1. Create a source file that specifies the information used in messages, message codes, and message symbols.
2. Use the MESSAGE command to compile this source file.
3. Link the resulting object module, either by itself or with another object module containing a program.
4. Run your program so that the messages are accessed, either directly or through the use of pointers.

See also the description of the Message utility in the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

13.11.3.1 Creating the Message Source File

A message source file contains definition statements and directives. The following source message file defines the error messages generated by the sample 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 *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

13.11.3.1.1 Specifying the Facility To specify the name and number of the facility for which you are defining the error messages, use the .FACILITY directive. For instance, the following .FACILITY directive specifies the facility (program) INCOME and a facility number of 1:

```
.FACILITY INCOME, 1
```

In addition to identifying the program associated with the error messages, the .FACILITY directive specifies the facility prefix that is added to each condition name to create the symbolic name used to reference the message. By default, the prefix is the facility name followed by an underscore. For example, a condition name BADFIXVAL defined following the previous .FACILITY directive is referenced as INCOME_BADFIXVAL. You can specify a prefix other than the specified program name by specifying the /PREFIX qualifier of the .FACILITY directive.

By convention, system-defined condition values are identified by the facility name, followed by a dollar sign ($), an underscore (_), and the condition name. User-defined condition values 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 values, 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. Digital recommends that you have one .FACILITY directive per message file.

13.11.3.1.2 Specifying the Severity Use the .SEVERITY directive and one of the following keywords to specify the severity of one or more condition values:

Success
Informational
Warning
Error
Severe or fatal

All condition values defined after a .SEVERITY directive have the specified severity (unless you use the /SEVERITY qualifier with 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 values 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
```

13.11.3.1.3 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"
```

13.11.3.1.4 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 *OpenVMS Command Definition, Librarian, and Message Utilities 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 13.11.4).

13.11.3.1.5 Compiling and Linking the Messages Use the DCL command MESSAGE to compile a message source file into an object module. The following command compiles the message source file INCMSG.MSG into an object module named INCMSG in the file INCMSG.OBJ:

```
$ MESSAGE INCMSG
```

To specify an object file name that is different from the source file name, use the /OBJECT qualifier of the MESSAGE command. To specify an object module name that is different from the source file name, use the .TITLE directive in the message source file.

13.11.3.1.6 Linking the Message Object Module The message object module must be linked with your program so the system can reference the messages. To simplify linking a program with the message object module, include the message object module in the program's object library. For example, to include the message module in INCOME's object library, enter the following:

```
$ LIBRARY INCOME.OLB INCMSG.OBJ
```

13.11.3.1.7 Accessing the Message Object Module from Multiple Programs Including the message module in the program's object library does not allow other programs-access to the module's condition values 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
```

13.11.3.1.8 Modifying a Message Source Module To modify a message in the message library, modify and recompile the message source file, and then replace the module in the object module library. To access the modified messages, a program must relink against the object module library (or the message object module). The following command enters the module INCMSG into the message library MESSAGLIB; if MESSAGLIB already contains an INCMSG module, it is replaced:

```
$ LIBRARY/REPLACE MESSAGLIB INCMSG
```

13.11.3.1.9 Accessing Modified Messages Without Relinking To allow a program to access modified messages without relinking, create a message pointer file. Message pointer files are useful if you need to provide messages in more than one language or frequently change the text of existing messages. See the description of the Message utility in the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

13.11.4 Signaling User-Defined Values and Messages with Global and Local Symbols

To signal a user-defined condition value, you use the symbol formed by the facility prefix and the condition name (for example, INCOME__BADFIXVAL). Typically, you reference a condition value as a global symbol; however, you can create an include file (similar to the modules in the system library SYS$LIBRARY:FORSTSDEF.TLB) to define the condition values as local symbols. If the message text contains FAO arguments, you must specify parameters for those arguments when you signal the condition value.

13.11.4.1 Signaling with Global Symbols

To signal a user-defined condition value using a global symbol, declare the appropriate condition value in the appropriate section of the program unit, and then invoke the RTL routine LIB$SIGNAL to signal the condition value. The following statements signal the condition value 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
```

13.11.4.2 Signaling with Local Symbols

To signal a user-defined condition value using a local symbol, you must first create a file containing PARAMETER statements that equate each condition value with its user-defined condition value. To create such a file, do the following:

1. Create a listing file—Compile the message source file with the /LIST qualifier to 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 column is the value of the condition value, 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 value as a longword variable, and use a language statement to equate the condition value to its hexadecimal condition value.

   Assuming a prefix of INCOME__, editing the previous statement results in the following statements:

   ```
   INTEGER INCOME__NOHOUSE
   PARAMETER (INCOME__NOHOUSE = '08018020'X)
   ```

3. Rename the listing file—Name the edited listing file using the same name as the source file and a file type for your programming language (for example, .FOR for DEC Fortran).

In the definition section of your program unit, declare the local symbol definitions by naming your edited listing file in an INCLUDE statement. (You must still link the message object file with your program.) Invoke the RTL routine LIB$SIGNAL to signal the condition code. The following statements signal the condition code INCOME__NOHOUSE when the value of FIX_HOUSE_NO is less than 1 or greater than the value of TOTAL_HOUSES:

```
! Specify the full file specification
INCLUDE '$DISK1:[DEV.INCOME]INCMSG.FOR'
   .
   .
   .
IF ((FIX_HOUSE_NO .GT. TOTAL_HOUSES) .OR.
2    FIX_HOUSE_NO .LT. 1)) THEN
  CALL LIB$SIGNAL (%VAL (INCOME__NOHOUSE))
END IF
```

13.11.4.3 Specifying FAO Parameters

If the message contains FAO arguments, you must specify the number of FAO arguments as the second argument of LIB$SIGNAL, the first FAO argument as the third argument, the second FAO argument as the fourth argument, and so on. Pass string FAO arguments by descriptor (the default). For example, to signal the condition code INCOME__NONUMBER, where FIX_HOUSE_NO contains the erroneous house number, specify the following:

```
EXTERNAL INCOME__NONUMBER
   .
   .
   .
IF ((FIX_HOUSE_NO .GT. TOTAL_HOUSES) .OR.
2    FIX_HOUSE_NO .LT. 1)) THEN
  CALL LIB$SIGNAL (%VAL (%LOC (INCOME__NONUMBER)),
2                  %VAL (1),
2                  %VAL (FIX_HOUSE_NO))
  END IF
```

To signal the condition code NOFILE, where FILE_NAME contains the invalid file specification, specify the following:

```
EXTERNAL INCOME__NOFILE
   .
   .
   .
IF (IOSTAT .EQ. FOR$IOS_FILNOTFOU)
2  CALL LIB$SIGNAL (%VAL (%LOC (INCOME__NOFILE)),
2                   %VAL (1),
2                   FILE_NAME)
```

Both of the previous examples use global symbols for the condition values. Alternatively, you could use local symbols, as described in Section 13.11.4.2.

13.12 Writing a Condition Handler

When you write a condition handler into your program, the process involves one or more of the following actions:

- Establish the handler in the stack frame of your routine.
- Write a condition-handling routine, or choose one of the run-time library routines that handles exception conditions.
- Include a call to a run-time library signal-generating routine.
- Use the Message utility to define your own exception conditions.
- Include a call to the SYS$PUTMSG system service to modify or log the system error message.

You can write a condition handler to take action when an exception condition is signaled. When the exception condition occurs, the OpenVMS Condition Handling facility sets up the signal argument vector and mechanism argument vector and begins the search for a condition handler. Therefore, your condition-handling routine must declare variables to contain the two argument vectors. Further, the handler must indicate the action to be taken when it returns to the OpenVMS Condition Handling facility. The handler uses its function value to do this. Thus, the calling sequence for your condition handler has the following format:

handler signal-args ,mechanism-args

signal-args
The address of a vector of longwords indicating the nature of the condition. See Section 13.8.2 for a detailed description.

mechanism-args
The address of a vector of longwords that indicates the state of the process at the time of the signal. See Section 13.8.3 and Section 13.8.4 for more details.

result
A condition value. Success (bit <0> = 1) causes execution to continue at the PC; failure (bit <0> = 0) causes the condition to be resignaled. That is, the system resumes the search for other handlers. If the handler calls the Unwind (SYS$UNWIND) system service, the return value is ignored and the stack is unwound. (See Section 13.12.3.)

Handlers can modify the contents of either the **signal-args** vector or the **mechanism-args** vector.

In order to protect compiler optimization, a condition handler and any routines that it calls can reference only arguments that are explicitly passed to handlers. They cannot reference COMMON or other external storage, and they cannot reference local storage in the routine that established the handler unless the compiler considers the storage to be volatile. Compilers that do not adhere to this rule must ensure that any variables referenced by the handler are always kept in memory, not in a register.

As mentioned previously, a condition handler can take one of three actions:

- Continue execution
- Resignal the exception condition and resume the stack scanning operation
- Call SYS$UNWIND to unwind the call stack to an earlier frame

The sections that follow describe how to write condition handlers to perform these three operations.

13.12.1 Continuing Execution

To continue execution from the instruction following the signal, with no error messages or traceback, the handler returns with the function value SS$_CONTINUE (bit <0> = 1). If, however, the condition was signaled with a call to LIB$STOP, the SS$_CONTINUE return status causes an error message (Attempt To Continue From Stop), and the image exits. The only way to continue from a call to LIB$STOP is for the condition handler to request a stack unwind.

If execution is to continue after a hardware fault (such as a reserved operand fault), the condition handler must correct the cause of the condition before returning the function value SS$_CONTINUE or requesting a stack unwind. Otherwise, the instruction that caused the fault executed again.

VAX

Note

On most VAX systems, hardware floating-point traps have been changed to hardware faults. If you still want floating-point exception conditions to be treated as traps, use LIB$SIM_TRAP to simulate the action of floating-point traps. ♦

Alpha

On Alpha systems, LIB$SIM_TRAP is not supported. Table 13–4 lists the run-time library routines that are supported and not supported on Alpha systems. ♦

13.12.2 Resignaling

Condition handlers check for specific errors. If the signaled condition is not one of the expected errors, the handler resignals. That is, it returns control to the OpenVMS Condition Handling facility with the function value SS$_RESIGNAL (with bit <0> clear). To alter the severity of the signal, the handler modifies the low-order 3 bits of the condition value and resignals.

For an example of resignaling, see Section 13.8.5.

13.12.3 Unwinding the Call Stack

A condition handler can dismiss the signal by calling the system service SYS$UNWIND. The stack unwind is initiated when a condition handler that has called SYS$UNWIND returns to OpenVMS Condition Handling facility. For an explanation of unwinding, see Section 13.10.1; for an example of using SYS$UNWIND to return control to the program, see Section 13.12.4.5.

13.12.4 Example of Writing a Condition Handler

The operating system passes two arrays to a condition handler. Any condition handler that you write should declare two dummy arguments as variable-length arrays, as in the following:

```
INTEGER*4 FUNCTION HANDLER (SIGARGS,
2                           MECHARGS)

INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
   .
   .
   .
```

13.12.4.1 Signal Array

The first dummy argument, the signal array, describes the signaled condition codes that indicate which error occurred and the state of the process when the condition code was signaled. For the structure of the signal array, see Section 13.8.2.

13.12.4.2 Mechanism Array

The second dummy argument, the mechanism array, describes the state of the process when the condition code was signaled. Typically, a condition handler references only the call depth and the saved function value. Currently, the mechanism array contains exactly five elements; however, because its length is subject to change, you should declare the dummy argument as a variable-length array. For the structure of the mechanism array, see Section 13.8.3.

Usually you write a condition handler in anticipation of a particular set of condition values. Because a handler is invoked in response to any signaled condition code, 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, resignal the condition code by equating the function value of the handler to the global symbol SS$_RESIGNAL.

13.12.4.3 Comparing the Signaled Condition with an Expected Condition

You can use the RTL routine LIB$MATCH_COND to compare the signaled condition code to a list of expected condition values. The first argument passed to LIB$MATCH_COND is the signaled condition code, the second element of the signal array. The rest of the arguments passed to LIB$MATCH_COND are the expected condition values. LIB$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 DEC Fortran I/O errors. If it is not, the condition handler resignals the condition code. Note that, when a DEC Fortran I/O error is signaled, the signal array describes operating system's condition code, not the DEC 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
```

13.12.4.4 Exiting from the Condition Handler

You can exit from a condition handler in one of three ways:

- Continue execution of the program—If you equate the function value of the condition handler to SS$_CONTINUE, the condition handler returns control to the program at the statement that signaled the condition (fault) or the statement following the one that signaled the condition (trap). The RTL routine LIB$SIGNAL generates a trap so that control is returned to the statement following the call to LIB$SIGNAL.

  In the following example, if the condition code is one of the expected codes, the condition handler displays a message and then returns the value SS$_CONTINUE to resume program execution. (Section 13.11 describes how to display a message.)

  ```
  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 operating system to execute the next condition handler. If you modify the signal array or mechanism array before resignaling, the modified arrays are passed to the next condition handler.

  In the following example, if the condition code is not one of the expected codes, the handler resignals:

  ```
  INDEX = LIB$MATCH_COND (SIGARGS(2),
  2                       FOR$_FILNOTFOU,
  2                       FOR$_OPEFAI,
  2                       FOR$_NO_SUCDEV,
  2                       FOR$_FILNAMSPE)
  ```

  ```
  IF (INDEX .EQ. 0) THEN
    HANDLER = SS$_RESIGNAL
  END IF
  ```

- Continue execution of the program at a previous location—If you call the SYS$UNWIND system service, the condition handler can return control to any point in the program unit that incurred the exception, the program unit that invoked the program unit that incurred the exception, and so on back to the program unit that established the condition handler.

13.12.4.5 Returning Control to the Program

Your handlers should return control either to the program unit that established the handler or to the program unit that invoked the program unit that established the handler.

To return control to the program unit that established the handler, invoke SYS$UNWIND and pass the call depth (third element of the mechanism array) as the first argument with no second argument.

```
! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
   .
   .
   .
CALL SYS$UNWIND (MECHARGS(3),)
```

To return control to the caller of the program unit that established the handler, invoke SYS$UNWIND without passing any arguments.

```
! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
   .
   .
   .
CALL SYS$UNWIND (,)
```

The first argument SYS$UNWIND specifies the number of program units to unwind (remove from the stack). If you specify this argument at all, you should do so as shown in the previous example. (MECHARGS(3) contains the number of program units that must be unwound to reach the program unit that established the handler that invoked SYS$UNWIND.) The second argument SYS$UNWIND 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$UNWIND removes a program unit from the stack, it invokes any condition handler established by that program unit and passes the condition handler the SS$_UNWIND condition code. To prevent the condition handler from resignaling the SS$_UNWIND condition code (and so complicating the unwind operation), include SS$_UNWIND as an expected condition code when you invoke LIB$MATCH_COND. When the condition code is SS$_UNWIND, your condition handler might perform necessary cleanup operations or do nothing.

In the following example, if the condition code is SS$_UNWIND, no action is performed. If the condition code is another of the expected codes, the handler displays the message and then returns control to the program unit that called the program unit that established the condition handler.

```
INTEGER FUNCTION HANDLER (SIGARGS,
2                         MECHARGS)

! Declare dummy arguments
INTEGER*4 SIGARGS(*),
2         MECHARGS(*)
INCLUDE '($FORDEF)'
INCLUDE '($SSDEF)'
INTEGER*4 INDEX,
2         LIB$MATCH_COND
INDEX = LIB$MATCH_COND (SIGARGS(2),
2                       SS$_UNWIND,
2                       FOR$_FILNOTFOU,
2                       FOR$_OPEFAI,
2                       FOR$_NO_SUCDEV,
2                       FOR$_FILNAMSPE)
IF (INDEX .EQ. 0) THEN
  ! Unexpected condition, resignal
  HANDLER = SS$_RESIGNAL
ELSE IF (INDEX .EQ. 1) THEN
  ! Unwinding, do nothing
ELSE IF (INDEX .GT. 1) THEN
   .
   .
   .
              ! Display the message
   .
   .
   .
  CALL SYS$UNWIND (,)
END IF
```

13.12.5 Example of Condition-Handling Routines

The following example shows two procedures, A and B, that have declared condition handlers. The notes describe the sequence of events that would occur if a call to a system service failed during the execution of procedure B.

```
/* PGMA */

#include <stdio.h>
#include <ssdef.h>

unsigned int sigargs[],mechargs[];

main() {
        unsigned int status, vector=0, old_handler;

        old_handler = LIB$ESTABLISH( handlera );              1

        status = pgmb (arglst);                               2
   .
   .
   .
}

/* PGMB */

#include <stdio.h>
#include <ssdef.h>

main() {

 old_handler = LIB$ESTABLISH( handlerb );                     3
   .
   .
   .

}                                                             4

/* Handler A */                                               5

int handlera( sigargs, mechargs ) {

/* Compare condition value signalled with expected value */
                                                              6
                        if (sigargs[2] != SS$_SSFAIL)
                                goto no_fail;
   .
   .
   .
/* Signal to continue */

                        return SS$_CONTINUE;

/* Signal to resignal */
no_fail:
                        return SS$_RESIGNAL;

}

/* Handler B */

int handlerb( sigargs, mechargs ) {

/* Compare condition value signalled with expected value */
                        if (sigargs[2] != SS$_BREAK)             7
                                goto no_fail;
   .
   .
   .
                        return SS$_CONTINUE;
no_fail:
                        return SS$_RESIGNAL;

}
```

1 Procedure A executes and establishes condition handler HANDLERA. HANDLERA is set up to respond to exceptions caused by failures in system service calls.

2 During its execution, procedure A calls procedure B.

3 Procedure B establishes condition handler HANDLERB. HANDLERB is set up to respond to breakpoint faults.

4 While procedure B is executing, an exception occurs caused by a system service failure.

5 The dispatcher returns control to procedure B, and execution of procedure B resumes at the instruction following the system service failure.

6 The exception dispatcher resumes its search for a condition handler and calls HANDLERA.

7 HANDLERA handles the system service failure exception, corrects the condition, places the return value SS$_CONTINUE in R0, and returns control to the exception dispatcher.

13.13 Debugging a Condition Handler

You can debug a condition handler as you would any subprogram, except that you cannot use the DEBUG command STEP/INTO to enter a condition handler. You must set a breakpoint in the handler and wait for the debugger to invoke the handler.

Typically, to trace execution of a condition handler, you set breakpoints at the statement in your program that should signal the condition code, at the statement following the one that should signal, and at the first executable statement in your condition handler.

13.14 Run-Time Library Condition-Handling Routines

The run-time library provides several routines that can be established as condition handlers or called from a condition handler to handle signaled exception conditions. This section shows how to use these routines.

13.14.1 Converting a Floating-Point Fault to a Floating-Point Trap (VAX Only)

VAX

On VAX systems, a trap is an exception condition that is signaled after the instruction that caused it has finished executing. A fault is an exception condition that is signaled during the execution of the instruction. When a trap is signaled, the program counter (PC) in the signal argument vector points to the next instruction after the one that caused the exception condition. When a fault is signaled, the PC in the signal argument vector points to the instruction that caused the exception condition. See the *VAX Architecture Reference Manual* for more information about faults and traps.

LIB$SIM_TRAP can be established as a condition handler or be called from a condition handler to convert a floating-point fault to a floating-point trap. After LIB$SIM_TRAP is called, the PC points to the instruction after the one that caused the exception condition. Thus, your program can continue execution without fixing up the original condition. LIB$SIM_TRAP intercepts only floating-point overflow, floating-point underflow, and divide-by-zero faults. ♦

13.14.2 Changing a Signal to a Return Status

When it is preferable to detect errors by signaling but the calling routine expects a returned status, LIB$SIG_TO_RET can be used by the routine that signals. For instance, if you expect a particular condition code to be signaled, you can prevent the operating system from invoking the default condition handler by establishing a different condition handler. LIB$SIG_TO_RET is a condition handler that converts any signaled condition to a return status. The status is returned to the caller of the routine that established LIB$SIG_TO_RET. You may establish LIB$SIG_TO_RET as a condition handler by specifying it in a call to LIB$ESTABLISH.

Alpha On Alpha systems, LIB$ESTABLISH is not supported, though high-level languages may support it for compatibility. ♦

LIB$SIG_TO_RET can also be called from another condition handler. If LIB$SIG_TO_RET is called from a condition handler, the signaled condition is returned as a function value to the caller of the establisher of that handler when the handler returns to the OpenVMS Condition Handling facility. When a signaled exception condition occurs, LIB$SIG_TO_RET routine does the following:

- Places the signaled condition value in the image of R0 that is saved as part of the mechanism argument vector.
- Calls the Unwind (SYS$UNWIND) system service with the default arguments. After returning from LIB$SIG_TO_RET (when it is established as a condition handler) or after returning from the condition handler that called LIB$SIG_TO_RET (when LIB$SIG_TO_RET is called from a condition handler), the stack unwinds to the caller of the routine that established the handler.

Your calling routine can now test R0, as if the called routine had returned a status, and specify an error recovery action.

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.

To change a signal to a return status, you must put any code that might signal a condition code into a function where the function value is a return status. The function containing the code must perform the following operations:

- Declare LIB$SIG_TO_RET—Declare the condition handler LIB$SIG_TO_RET.
- Establish LIB$SIG_TO_RET—Invoke the run-time library procedure LIB$ESTABLISH to establish a condition handler for the current program unit. Specify the name of the condition handler LIB$SIG_TO_RET as the only argument.
- Initialize the function value—Initialize the function value to SS$_NORMAL so that, if no condition code is signaled, the function returns a success status to the invoking program unit.

- Declare necessary dummy arguments—If any statement that might signal a condition code is a subprogram that requires dummy arguments, pass the necessary arguments to the function. In the function, declare each dummy argument exactly as it is declared in the subprogram that requires it and specify the dummy arguments in the subprogram invocation.

If the program unit GET_1_STAT in the following function signals a condition code, LIB$SIG_TO_RET changes the signal to the return status of the INTERCEPT_SIGNAL function and returns control to the program unit that invoked INTERCEPT_SIGNAL. (If GET_1_STAT has a condition handler established, the operating system invokes that handler before invoking LIB$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 13.5.1) to determine which condition code, if any, was signaled during execution of INTERCEPT_SIGNAL.

13.14.3 Changing a Signal to a Stop

LIB$SIG_TO_STOP causes a signal to appear as though it had been signaled by a call to LIB$STOP.

LIB$SIG_TO_STOP can be enabled as a condition handler for a routine or be called from a condition handler. When a signal is generated by LIB$STOP, the severity code is forced to severe, and control cannot return to the routine that signaled the condition. See Section 13.12.1 for a description of continuing normal execution after a signal.

13.14.4 Matching Condition Values

LIB$MATCH_COND checks for a match between two condition values to allow a program to branch according to the condition found. If no match is found, the routine returns zero. The routine matches only the condition identification field (STS$V_COND_ID) of the condition value; it ignores the control bits and the severity field. If the facility-specific bit (STS$V_FAC_SP = bit <15>) is clear in **cond-val** (meaning that the condition value is systemwide), LIB$MATCH_COND ignores the facility code field (STS$V_FAC_NO = bits <27:17>) and compares only the STS$V_MSG_ID fields (bits <15:3>).

13.14.5 Correcting a Reserved Operand Condition (VAX Only)

VAX

On VAX systems, after a signal of SS$_ROPRAND during a floating-point instruction, LIB$FIXUP_FLT finds the operand and changes it from –0.0 to a new value or to +0.0. ♦

13.14.6 Decoding the Instruction That Generated a Fault (VAX Only)

VAX

On VAX systems, LIB$DECODE_FAULT locates the operands for an instruction that caused a fault and passes the information to a user action routine. When called from a condition handler, LIB$DECODE_FAULT locates all the operands and calls an action routine that you supply. Your action routine performs the steps necessary to handle the exception condition and returns control to LIB$DECODE_FAULT. LIB$DECODE_FAULT then restores the operands and the environment, as modified by the action routine, and continues execution of the instruction. ♦

13.15 Exit Handlers

When an image exits, the 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 by user-defined exit handlers.
- Executes a number of cleanup operations collectively known as image rundown. The following is a list of some of these cleanup operations:
  - Canceling outstanding ASTs and timer requests.
  - Deassigning any channel assigned by your program and not already deassigned by your program or the system.
  - Deallocating devices allocated by the program.
  - Dissociating common event flag clusters associated with the program.
  - Deleting user-mode logical names created by the program. (Unless you specify otherwise, logical names created by SYS$CRELNM are user-mode logical names.)
  - Restoring internal storage (for example, stacks or mapped sections) to its original state.

If any exit handler exits using the EXIT (SYS$EXIT) system service, none of the remaining handlers is executed. In addition, if an image is aborted by the DCL command STOP (the user presses Ctrl/Y and then enters STOP), the system performs image rundown and does not invoke any exit handlers. Like the DCL STOP/ID, SYS$DELPRC bypasses all exit handlers, except the rundown specified in the privileged library vector (PLV) privileged shareable images, and deletes the process. (The DCL command EXIT invokes the exit handlers before running down the image.)

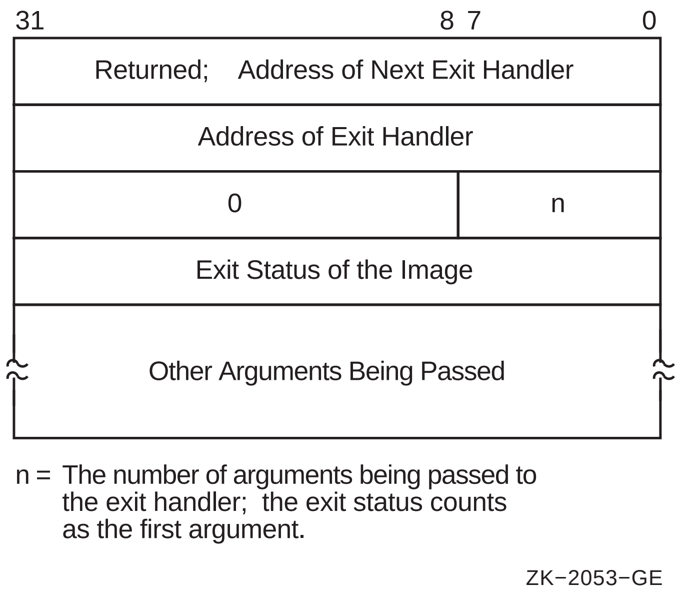
A hang-up to a terminal line causes DCL to delete the master process's subprocesses. However, if the subprocesses's exit handler is in a main image installed with privilege, then that exit handler is run even with the DCL command STOP. Also, if the subprocess was spawned NOWAIT, then the spawning process's exit handler is run as well.

Use exit handlers to perform any cleanup that your program requires in addition to the normal rundown operations performed by the 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.

13.15.1 Establishing an Exit Handler

To establish an exit handler, use the SYS$DCLEXH system service. The SYS$DCLEXH system service requires one argument—a variable-length data structure that describes the exit handler. Figure 13–16 illustrates the structure of an exit handler.

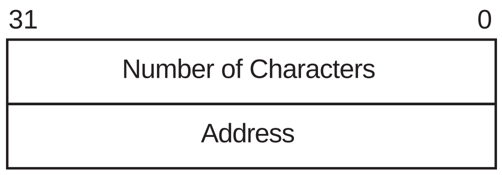
Figure 13–16 Structure of an Exit Handler


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−GE

The first longword of the structure contains the address of the next handler. The 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, do not supply the final status value; when the operating system invokes an exit handler, it passes the handler the final status value of the exiting program.

To pass an argument with a numeric data type, use programming language statements to assign the address of a numeric variable to one of the longwords in the exit-handler data structure. To pass an argument with a character data type, create a descriptor of the following form:



ZK−2054−GE

Use the language statements to assign the address of the descriptor to one of the longwords in the exit-handler data structure.

The following program segment establishes an exit handler with two arguments, the mandatory status argument and a character argument:

```
   .
   .
   .
! Arguments for exit handler
INTEGER EXIT_STATUS       ! Status
CHARACTER*12 STRING       ! String
STRUCTURE /DESCRIPTOR/
  INTEGER SIZE,
2        ADDRESS
END STRUCTURE
RECORD /DESCRIPTOR/ EXIT_STRING
! Setup for exit handler
STRUCTURE /EXIT_DESCRIPTOR/
 INTEGER LINK,
2       ADDR,
2       ARGS /2/,
2       STATUS_ADDR,
2       STRING_ADDR
END STRUCTURE
RECORD /EXIT_DESCRIPTOR/ HANDLER
! Exit handler
EXTERNAL EXIT_HANDLER
   .
   .
   .
! Set up descriptor
EXIT_STRING.SIZE = 12     ! Pass entire string
EXIT_STRING.ADDRESS = %LOC (STRING)
! Enter the handler and argument addresses
! into the exit handler description
HANDLER.ADDR = %LOC(EXIT_HANDLER)
HANDLER.STATUS_ADDR = %LOC(EXIT_STATUS)
HANDLER.STRING_ADDR = %LOC(EXIT_STRING)
! Establish the exit handler
CALL SYS$DCLEXH (HANDLER)
   .
   .
   .
```

An exit handler can be established at any time during your program and remains in effect until it is canceled (with SYS$CANEXH) or executed. If you establish more than one handler, the handlers are executed in reverse order: the handler established last is executed first; the handler established first is executed last.

13.15.2 Writing an Exit Handler

Write an exit handler as a subroutine, because 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.

In the following example, 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. Because it is important that each program clear its flag before exiting, you create an exit handler 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. In this example, since the cleanup operation is to be performed regardless of whether the program completes successfully, the final status is not examined in the exit routine. (This subroutine would not be used with the exit handler declaration in the previous example.)

CLEAR_FLAG.FOR

```
SUBROUTINE CLEAR_FLAG (EXIT_STATUS,
2                      FLAG)
! Exit handler clears the event flag

! Declare dummy argument
INTEGER EXIT_STATUS,
2       FLAG
! Declare status variable and system routine
INTEGER STATUS,
2       SYS$ASCEFC,
2       SYS$CLREF
! Associate with the common event flag
! cluster and clear the flag
STATUS = SYS$ASCEFC (%VAL(FLAG),
2                    'ALIVE',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))

END
```

If for any reason you must perform terminal I/O from an exit handler, use appropriate RTL routines. Trying to access the terminal from an exit handler using language I/O statements may cause a redundant I/O error.

13.15.3 Debugging an Exit Handler

To debug an exit handler, you must set a breakpoint in the handler and wait for the operating system to invoke that handler; you cannot use the DEBUG command STEP/INTO to enter an exit handler. In addition, when the debugger is invoked, it establishes an exit handler that exits using the SYS$EXIT system service. If you invoke the debugger when you invoke your image, the debugger's exit handler does not affect your program's handlers because the debugger's handler is established first and so executes last. However, if you invoke the debugger after your program begins executing (the user presses Ctrl/Y and then types DEBUG), the debugger's handler may affect the execution of your program's exit handlers, because one or more of your handlers may have been established before the debugger's handler and so is not executed.

13.15.4 Example of an Exit Handler

As in the example in Section 13.15.2, write the exit handler as a subroutine because 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 specify in the call to SYS$DCLEXH.

In the following example, assume that two or more programs are cooperating. 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. Because each program must clear its flag before exiting, you create an exit handler 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.

In the following example, because the cleanup operation is to be performed regardless of whether the program completes successfully, the final status is not examined in the exit routine.

```
! Arguments for exit handler
INTEGER*4 EXIT_STATUS       ! Status
INTEGER*4 FLAG /64/
! Setup for exit handler
STRUCTURE /EXIT_DESCRIPTOR/
 INTEGER LINK,
2        ADDR,
2        ARGS /2/,
2        STATUS_ADDR,
2        FLAG_ADDR
END STRUCTURE
RECORD /EXIT_DESCRIPTOR/ HANDLER

! Exit handler
EXTERNAL EXIT_HANDLER

INTEGER*4 STATUS,
2         SYS$ASCEFC,
2         SYS$SETEF

! Associate with the common event flag
! cluster and set the flag.
STATUS = SYS$ASCEFC (%VAL(FLAG),
2                    'ALIVE',,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
STATUS = SYS$SETEF (%VAL(FLAG))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL (STATUS))
! Do not exit until cooperating program has a chance to
! associate with the common event flag cluster.

! Enter the handler and argument addresses
! into the exit handler description.
HANDLER.ADDR = %LOC(EXIT_HANDLER)
HANDLER.STATUS_ADDR = %LOC(EXIT_STATUS)
HANDLER.FLAG_ADDR = %LOC(FLAG)
! Establish the exit handler.
CALL SYS$DCLEXH (HANDLER)

! Continue with program
              .
              .
              .
END

! Exit Subroutine
```

```
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))
```

14
Synchronizing Data Access and Program Operations

This chapter describes the operating system's synchronization features. It focuses on how the operating system synchronizes the sequencing of events to perform memory operations. Memory access synchronization techniques are based on synchronization techniques that other types of storage use, whether to share hardware resources or data in files.

This chapter contains the following sections:

Section 14.1 describes synchronization, execution of threads, and atomicity.

Section 14.2 describes alignment, granularity, ordering of read and write operations, and performing memory read and write operations on VAX and Alpha systems in uniprocessor and multiprocessor environments.

Section 14.3 describes how to perform memory read-modify-write operations on VAX and Alpha systems in uniprocessor and multiprocessor environments.

Section 14.4 describes hardware-level synchronization methods, such as interrupt priority level, load-locked/store-conditional and interlocked instructions, memory barriers, and PALcode routines.

Section 14.5 describes software-level synchronization methods, such as process-private synchronization techniques, process priority, and spin locks. It also describes how to write applications for a multiprocessor environment using higher-level synchronization methods and how to write to global sections.

Section 14.6 describes how to use local and common event flags for synchronization.

Section 14.7 describes how to use the PPL$ routines for synchronization in a multiprocessor configuration.

Section 14.8 describes how to use SYS$SYNCH system service for synchronization.

14.1 Overview of Synchronization

Software synchronization refers to the coordination of events in such a way that only one event happens at a time. This kind of synchronization is a serialization or sequencing of events. Serialized events are assigned an order and processed one at a time in that order. While a serialized event is being processed, no other event in the series is allowed to disrupt it.

By imposing order on events, synchronization allows reading and writing of several data items indivisibly, or atomically, in order to obtain a consistent set of data. For example, all of process A's writes to shared data must happen before or after process B's writes or reads, but not during process B's writes or reads. In this case, all of process A's writes must happen indivisibly for the operation to be correct. This includes process A's updates—reading of a data item, modifying it, and writing it back (read-modify-write sequence). Other synchronization techniques are used to ensure the completion of an asynchronous system service before the caller tries to use the results of the service.

14.1.1 Threads of Execution

Code threads that can execute within a process include the following:

- Mainline code in an image being executed by a kernel thread, or multiple threads
- Asynchronous system traps (ASTs) that interrupt the image
- Condition handlers established by the image and that run after exceptions occur
- Inner access-mode threads of execution that run as a result of system service, OpenVMS Record Management Services (RMS), and command language interpreter (CLI) callback requests

Process-based threads of execution can share any data in the P0 and P1 address space and must synchronize access to any data they share. A thread of execution can incur an exception, which results in passing of control to a condition handler. Alternatively, the thread can receive an AST, which results in passing of control to an AST procedure. Further, an AST procedure can incur an exception, and a condition handler's execution can be interrupted by an AST delivery. If a thread of execution requests a system service or RMS service, control passes to an inner access-mode thread of execution. Code that executes in the inner mode can also incur exceptions, receive ASTs, and request services.

Multiple processes, each with its own set of threads of execution, can execute concurrently. Although each process has private P0 and P1 address space, processes can share data in a global section mapped into each process's address spaces. You need to synchronize access to global section data because a thread of execution accessing the data in one process can be rescheduled, allowing a thread of execution in another process to access the same data before the first process completes its work. Although processes access the same system address space, the protection on system space pages usually prevents outer mode access. However, process-based code threads running in inner access modes can access data concurrently in system space and must synchronize access to it.

Interrupt service routines access only system space. They must synchronize access to shared system space data among themselves and with process-based threads of execution.

A CPU-based thread of execution and an I/O processor must synchronize access to shared data structures, such as structures that contain descriptions of I/O operations to be performed.

Multiprocessor execution increases synchronization requirements when the threads that must synchronize can run concurrently on different processors. Because a process executes on only one processor at a time, synchronization of threads of execution within a process is unaffected by whether the process runs on a uniprocessor or on a symmetric multiprocessing (SMP) system. However, multiple processes can execute simultaneously on different processors. Because of this, processes sharing data in a global section can require additional synchronization for SMP system execution. Further, process-based inner mode and interrupt-based threads can execute simultaneously on different processors and can require synchronization of access to system space beyond what is sufficient on a uniprocessor.

14.1.2 Atomicity

Atomicity is a type of serialization that refers to the indivisibility of a small number of actions, such as those occurring during the execution of a single instruction or a small number of instructions. With more than one action, no single action can occur by itself. If one action occurs, then all the actions occur. Atomicity must be qualified by the viewpoint from which the actions appear indivisible: an operation that is atomic for threads running on the same processor can appear as multiple actions to a thread of execution running on a different processor.

An atomic memory reference results in one indivisible read or write of a data item in memory. No other access to any part of that data can occur during the course of the atomic reference. Atomic memory references are important for synchronizing access to a data item that is shared by multiple writers or by one writer and multiple readers. References need not be atomic to a data item that is not shared or to one that is shared but is only read.

14.2 Memory Read and Memory Write Operations

This section presents the important concepts of **alignment** and **granularity** and how they affect the access of shared data on VAX and Alpha systems. It also discusses the importance of the order of reads and writes are completed on VAX and Alpha systems, and how VAX and Alpha system perform memory reads and writes.

14.2.1 Alignment

The term **alignment** refers to the placement of a data item in memory. For a data item to be naturally aligned, its lowest-addressed byte must reside at an address that is a multiple of the size of the data item in bytes. For example, a naturally aligned longword has an address that is a multiple of 4. The term **naturally aligned** is usually shortened to “aligned.”

VAX

On VAX systems, a thread on a VAX uniprocessor or multiprocessor can read and write aligned byte, word, and longword data atomically with respect to other threads of execution accessing the same data. ♦

Alpha

On Alpha systems, in contrast to the variety of memory accesses allowed on VAX systems, an Alpha processor allows atomic access only to an aligned longword or an aligned quadword. Reading or writing an aligned longword or quadword of memory is atomic with respect to any other thread of execution on the same processor or on other processors. ♦

14.2.2 Granularity

VAX and Alpha systems differ in granularity of data access. The phrase **granularity of data access** refers to the size of neighboring units of memory that can be written independently and atomically by multiple processors. Regardless of the order in which the two units are written, the results must be identical.

VAX

VAX systems have byte granularity: individual adjacent or neighboring bytes within the same longword can be written by multiple threads of execution on one or more processors, as can aligned words and longwords.

VAX systems provide instructions that can manipulate byte-sized and aligned word-sized memory data in a single, noninterruptible operation. On VAX systems, a byte-sized or word-sized data item that is shared can be manipulated individually. ♦

Alpha

Alpha systems have longword and quadword granularity. That is, only adjacent aligned longwords or quadwords can be written independently. Because Alpha systems support only instructions that load or store longword-sized and quadword-sized memory data, the manipulation of byte-sized and word-sized data on Alpha systems requires that the entire longword or quadword that contain the byte- or word-sized item to be manipulated. Thus, simply because of its proximity to an explicitly shared data item, neighboring data might become shared unintentionally. Manipulation of byte-sized and word-sized data on Alpha systems requires multiple instructions that:

1. Fetch the longword or quadword that contains the byte or word
2. Mask the nontargeted bytes
3. Manipulate the target byte or word
4. Store the entire longword or quadword

Because this sequence is interruptible, operations on byte and word data are not atomic on Alpha sytems. Also, this change in the granularity of memory access can affect the determination of which data is actually shared when a byte or word is accessed.

The absence of byte and word granularity on Alpha systems has important implications for access to shared data. In effect, any memory write of a data item other than an aligned longword or quadword must be done as a multiple-instruction read-modify-write sequence. Also, because the amount of data read and written is an entire longword or quadword, programmers must ensure that all accesses to fields within the longword or quadword are synchronized with each other. ♦

14.2.3 Ordering of Read and Write Operations

On VAX uniprocessor and multiprocessor systems, write operations and read operations appear to occur in the same order in which you specify them from the viewpoint of all types of external threads of execution. Alpha uniprocessor systems also guarantee that read and write operations appear ordered for multiple threads of execution running within a single process or within multiple processes running on a uniprocessor.

Alpha

On Alpha multiprocessor systems, you must order reads and writes explicitly to ensure that they occur in a specific order from the viewpoint of threads of execution on other processors. To provide the necessary operating system primitives and compatibility with VAX systems, Alpha systems support instructions that impose an order on read and write operations. ♦

14.2.4 Memory Reads and Memory Writes

VAX

On VAX systems, most instructions that read or write memory are noninterruptible. A memory write done with a noninterruptible instruction is atomic from the viewpoint of other threads on the same CPU.

On a uniprocessor system, reads and writes of bytes, words, longwords, and quadwords are atomic with respect to any thread on the processor. On a multiprocessor, not all of those accesses are atomic with respect to any thread on any processor; only reads and writes of bytes, aligned words, and aligned longwords are atomic. Accessing unaligned data can require multiple operations. As a result, even though an unaligned longword is written with a noninterruptible instruction, if it requires multiple memory accesses, a thread on another CPU might see memory in an intermediate state. VAX systems do not guarantee multiprocessor atomic access to quadwords. ♦

Alpha

On Alpha systems, there is no instruction that performs multiple memory accesses. Each load or store instruction performs a maximum of one load from or one store to memory. A load can occur only from an aligned longword or quadword. A store can occur only to an aligned longword or quadword.

Although reads and writes from one thread appear to occur ordered from the viewpoint of other threads on the same processor, there is no implicit ordering of reads and writes as seen by threads on other processors. ♦

14.3 Memory Read-Modify-Write Operations

A fundamental synchronization primitive for accessing shared data is an atomic read-modify-write operation. This operation consists of reading the contents of a memory location and replacing them with new contents based on the old. Any intermediate memory state is not visible to other threads. Both VAX systems and Alpha systems provide this synchronization primitive, but they implement it in significantly different ways.

14.3.1 Uniprocessor Operations

VAX

On VAX systems, many instructions are capable of performing a read-modify-write operation in a single, noninterruptible (atomic) sequence from the viewpoint of multiple application threads executing on a single processor. VAX systems provide synchronization among multiple threads of execution running on a uniprocessor system.

On VAX systems, the implicit dependence on the atomicity of VAX instructions is not recommended. Because of the optimizations they perform, the VAX compilers do not guarantee that a certain type of program statement, such as an increment operation (x=x+1), is implemented using a VAX atomic instruction, even if one exists. ♦

Alpha

On Alpha systems, there is no single instruction that performs an atomic read-modify-write operation. As a result, even uniprocessing applications in which processes access shared data must provide explicit synchronization of these accesses, usually through compiler semantics.

Read-modify-write operations that can be performed atomically on VAX systems require a sequence of instructions on Alpha systems. Because this sequence can be interrupted, the data may be left in an unstable state. For example, the VAX increment long (INCL) instruction fetches the contents of a specified longword, increments its value, and stores the value back in the longword, performing the operations without interruption. On Alpha systems, each step—fetching, incrementing, storing—must be explicitly performed by a separate instruction. Therefore, another thread in the process (for example, an AST routine) could execute before the sequence completes. However, because atomic updates are the basis of synchronization, and to provide compatibility with VAX systems, Alpha systems provide the following mechanisms to enable atomic read-modify-write updates:

- Privileged architecture library (PALcode) routines perform queue insertions and removals.
- Load-locked and store-conditional instructions create a sequence of instructions that implement an atomic update.

Note

The load-locked and store-conditional instructions also create a sequence of instructions that are atomic in a multiprocessor system. In contrast, a VAX INCL instruction is atomic only in a uniprocessor environment. ♦

14.3.2 Multiprocessor Operations

On multiprocessor systems, you must use special methods to ensure that a read-modify-write sequence is atomic. On VAX systems, interlocked instructions provide synchronization; on Alpha systems, load-locked and store-conditional instructions provide synchronization.

VAX

On VAX systems, a number of uninterruptible instructions are provided that both read and write memory with one instruction. When used with an operand type that is accessible in a single memory operation, each instruction provides an atomic read-modify-write sequence. The sequence is atomic with respect to threads of execution on the same VAX processor, but it is not atomic to threads on other processors. For instance, when a VAX CPU executes the instruction INCL *x*, it issues two separate commands to memory: a read, followed by a write of the incremented value. Another thread of execution running concurrently on another processor could issue a command to memory that reads or writes location *x* between the INCL's read and write. Section 14.4.3 describes read-modify-write sequences that are atomic with respect to threads on all VAX CPUs in an SMP system.

On a VAX multiprocessor system, an atomic update requires an interlock at the level of the memory subsystem. To perform that interlock, the VAX architecture provides a set of interlocked instructions that include Add Aligned Word Interlocked (ADAWI), Remove from Queue Head Interlocked (REMQHI), and Branch on Bit Set and Set Interlocked (BBSSI).

If you code in VAX MACRO, you use the assembler to generate whatever instructions you tell it. If you code in a high-level language, you cannot assume that the compiler will compile a particular language statement into a specific code sequence. That is, you must tell the compiler explicitly to generate an atomic update. For further information, see the documentation for your high-level language. ♦

Alpha

On Alpha systems, there is no single instruction that performs an atomic read-modify-write operation. An atomic read-modify-write operation is only possible through a sequence that includes load-locked and store-conditional instructions, (see Section 14.4.2). Use of these instructions provides a read-modify-write operation on data within one aligned longword or quadword that is atomic with respect to threads on all Alpha CPUs in an SMP system. ♦

14.4 Hardware-Level Synchronization

VAX

On VAX systems, the following features assist with synchronization at the hardware level:

- Atomic memory references
- Noninterruptible instructions
- Interrupt priority level (IPL)
- Interlocked memory accesses

On VAX systems, many read-modify-write instructions, including queue manipulation instructions, are noninterruptible. These instructions provide an atomic update capability on a uniprocessor. A kernel-mode code thread can block interrupt and process-based threads of execution by raising the IPL. Hence, it can execute a sequence of instructions atomically with respect to the blocked threads on a uniprocessor. Threads of execution that run on multiple processors of an SMP system synchronize access to shared data with read-modify-write instructions that interlock memory. ♦

Alpha

On Alpha systems, some of these mechanisms are present, while others have been implemented in PALcode routines.

Alpha processors provide several features to assist with synchronization. Even though all instructions that access memory are noninterruptible, no single one performs an atomic read-modify-write. A kernel-mode thread of execution can raise the IPL in order to block other threads on that processor while it performs a read-modify-write sequence or while it executes any other group of instructions. Code that runs in any access mode can execute a sequence of instructions that contains load-locked (LD*x*_L) and store-conditional (ST*x*_C) instructions to perform a read-modify-write sequence that appears atomic to other threads of execution. Memory barrier instructions order a CPU's memory reads and writes from the viewpoint of other CPUs and I/O processors. Other synchronization mechanisms are provided by PALcode routines. ♦

The sections that follow describe the features of interrupt priority level, load-locked (LD*x*_L) and store-conditional (ST*x*_C) instructions, memory barriers, interlocked instructions, and PALcode routines.

14.4.1 Interrupt Priority Level

The operating system in a uniprocessor system synchronizes access to systemwide data structures by requiring that all threads sharing data run at the highest-priority IPL of the highest-priority interrupt that causes any of them to execute. Thus, a thread's accessing of data cannot be interrupted by any other thread that accesses the same data.

The IPL is a processor-specific mechanism. Raising the IPL on one processor has no effect on another processor. You must use a different synchronization technique on SMP systems where code threads run concurrently on different CPUs that must have synchronized access to shared system data.

VAX

On VAX systems, the code threads that run concurrently on different processors synchronize through instructions that interlock memory in addition to raising the IPL. Memory interlocks also synchronize access to data shared by an I/O processor and a code thread. ♦

Alpha

On Alpha systems, access to a data structure that is shared either by executive code running concurrently on different CPUs or by an I/O processor and a code thread must be synchronized through a load-locked/store-conditional sequence. ♦

14.4.2 LD*x*_L and ST*x*_C Instructions

Alpha

Because Alpha systems do not provide a single instruction that both reads and writes memory or mechanism to interlock memory against other interlocked accesses, you must use other synchronization techniques. Alpha systems provide the load-locked/store-conditional mechanism that allows a sequence of instructions to perform an atomic read-modify-write operation.

Load-locked (LD*x*_L) and store-conditional (ST*x*_C) instructions guarantee atomicity that is functionally equivalent to that of VAX systems. The LD*x*_L and ST*x*_C instructions can be used only on aligned longwords or aligned quadwords. The LD*x*_L and ST*x*_C instructions do not provide atomicity by blocking access to shared data by competing threads. Instead, when the LD*x*_L instruction executes, a CPU-specific lock bit is set. Before the data can be stored, the CPU uses the ST*x*_C instruction to check the lock bit. If another thread has accessed the data item in the time since the load operation began, the lock bit is cleared and the store is not performed. Clearing the lock bit signals the code thread to retry the load operation. That is, a load-locked/store-conditional sequence tests the lock bit to see whether the store succeeded. If it did not succeed, the sequence branches back to the beginning to start over. This loop repeats until the data is untouched by other threads during the operation.

By using the LD*x*_L and ST*x*_C instructions together, you can construct a code sequence that performs an atomic read-modify-write operation to an aligned longword or quadword. Rather than blocking other threads' modifications of the target memory, the code sequence determines whether the memory locked by the LD*x*_L instruction could have been written by another thread during the sequence. If it is written, the sequence is repeated. If it is not written, the store is performed. If the store succeeds, the sequence is atomic with respect to other threads on the same processor and on other processors. The LD*x*_L and ST*x*_C instructions can execute in any access mode.

Traditional VAX usage is for interlocked instructions to be used for multiprocessor synchronization. On Alpha systems, LD*x*_L and ST*x*_C instructions implement interlocks and can be used for uniprocessor synchronization. To achieve protection similar to the VAX interlock protection, you need to use memory barriers along with the load-locked and store-conditional instructions.

Some Alpha system compilers make the LD*x*_L and ST*x*_C instruction mechanism explicitly available as language built-in functions. For example, DEC C on Alpha systems includes a set of built-in functions that provide for atomic addition and for logical AND and OR operations. Also, Alpha system compilers make the mechanism available implicitly, because they use the LD*x*_L and ST*x*_C instructions to access declared data as requiring atomic accesses in a language-specific way. ♦

14.4.3 Interlocked Instructions

VAX

On VAX systems, seven instructions interlock memory. A memory interlock enables a VAX CPU or I/O processor to make an atomic read-modify-write operation to a location in memory that is shared by multiple processors. The memory interlock is implemented at the level of the memory controller. On a VAX multiprocessor system, an interlocked instruction is the only way to perform an atomic read-modify-write on a shared piece of data. The seven interlock memory instructions are as follows:

- ADAWI—Add aligned word, interlocked
- BBCCI—Branch on bit clear and clear, interlocked
- BBSSI—Branch on bit set and set, interlocked
- INSQHI—Insert entry into queue at head, interlocked
- INSQTI—Insert entry into queue at tail, interlocked
- REMQHI—Remove entry from queue at head, interlocked
- REMQTI—Remove entry from queue at tail, interlocked

The VAX architecture interlock memory instructions are described in detail in the *VAX Architecture Reference Manual*.

The following description of the interlocked instruction mechanism assumes that the interlock is implemented by the memory controller and that the memory contents are fresh.

When a VAX CPU executes an interlocked instruction, it issues an interlock-read command to the memory controller. The memory controller sets an internal flag and responds with the requested data. While the flag is set, the memory controller stalls any subsequent interlock-read commands for the same aligned longword from other CPUs and I/O processors, even though it continues to process ordinary reads and writes. Because interlocked instructions are noninterruptible, they are atomic with respect to threads of execution on the same processor.

When the VAX processor that is executing the interlocked instruction issues a write-unlock command, the memory controller writes the modified data back and clears its internal flag. The memory interlock exists for the duration of only one instruction. Execution of an interlocked instruction includes paired interlock-read and write-unlock memory controller commands.

When you synchronize data with interlocks, you must make sure that all accessors of that data use them. This means that memory references of an interlocked instruction are atomic only with respect to other interlocked memory references.

On VAX systems, the granularity of the interlock depends on the type of VAX system. A given VAX implementation is free to implement a larger interlock granularity than that which is required by the set of interlocked instructions listed above. On some processors, for example, while an interlocked access to a location is in progress, interlocked access to any other location in memory is not allowed. ♦

14.4.4 Memory Barriers

Alpha

On Alpha systems, there are no implied memory barriers except those performed by the PALcode routines that emulate the interlocked queue instructions. Wherever necessary, you must insert explicit memory barriers into your code to impose an order on memory references. Memory barriers are required to ensure both the order in which other members of an SMP system or an I/O processor see writes to shared data and the order in which reads of shared data complete.

There are two types of memory barrier:

- The MB instruction
- The instruction memory barrier (IMB) PALcode routine

The MB instruction guarantees that all subsequent loads and stores do not access memory until after all previous loads and stores have accessed memory from the viewpoint of multiple threads of execution. Even in a multiprocessor system, all of the instruction's reads of one processor always return the data from the most recent writes by that processor, assuming no other processor has written to the location. Alpha compilers provide semantics for generating memory barriers when needed for specific operations on data items.

The instruction memory barrier (IMB) PALcode routine must be used after a modification to the instruction stream to flush prefetched instructions. In addition, it also provides the same ordering effects as the MB instruction.

Code that modifies the instruction stream must be changed to synchronize the old and new instruction streams properly. Use of an REI instruction to accomplish this does not work on OpenVMS Alpha systems.

If a kernel mode code sequence changes the expected instruction stream, it must issue an IMB instruction after changing the instruction stream and before the time the change is executed. For example, if a device driver stores an instruction sequence in an extension to the unit control block (UCB) and then transfers control there, it must issue an IMB instruction after storing the data in the UCB but before transferring control to the UCB data.

The MACRO-32 compiler for OpenVMS Alpha provides the EVAX_IMB built-in to insert explicitly an IMB instruction in the instruction stream. ♦

14.4.5 PALcode Routines

Alpha

Privileged architecture library (PALcode) routines include Alpha instructions that emulate VAX queue and interlocked queue instructions. PALcode executes in a special environment with interrupts blocked. This feature results in noninterruptible updates. A PALcode routine can perform the multiple memory reads and memory writes that insert or remove a queue element without interruption. ♦

14.5 Software-Level Synchronization

The operating system uses the synchronization primitives provided by the hardware as the basis for several different synchronization techniques. The following sections summarize the operating system's synchronization techniques available to application software.

14.5.1 Synchronization Within a Process

Alpha

On Alpha systems without kernel threads, only one thread of execution can execute within a process at a time, so synchronizaton of threads that execute simultaneously is not a concern. However, a delivery of an AST or the occurrence of an exception can intervene in a sequence of instructions in one thread of execution. Because these conditions can occur, application design must take into account the need for synchronization with condition handlers and AST procedures.

On Alpha systems, writing bytes or words or performing a read-modify-write operation requires a sequence of Alpha instructions. If the sequence incurs an exception or is interrupted by AST delivery or an exception, another process code thread can run. If that thread accesses the same data, it can read incompletely written data or cause data corruption. Aligning data on natural boundaries and unpacking word and byte data reduce this risk.

An application written in a language other than VAX MACRO must identify to the compiler data accessed by any combination of mainline code, AST procedures, and condition handlers to ensure that the compiler generates code that is atomic with respect to other threads. Also, data shared with other processes must be identified. ♦

With process-private data accessed from both AST and non-AST threads of execution, the non-AST thread can block AST delivery by using the Set AST Enable (SYS$SETAST) system service. If the code is running in kernel mode, it can also raise IPL to block AST delivery. The *Guide to Creating OpenVMS Modular Procedures* describes the concept of AST reentrancy.

On a uniprocessor or in a symmetric multiprocessing (SMP) system, access to multiple locations with a read or write instructions or with a read-modify-write sequence is not atomic on VAX and Alpha systems. Additional synchronization methods are required to control access to the data. See Section 14.5.4 and the sections following it, which describe the use of higher-level synchronization techniques.

14.5.2 Synchronization in Inner Mode

Alpha

With kernel threads, the system allows multiple execution contexts, or threads within a process, that all share the same address space to run simultaneously. The synchronization provided by the SCHED spinlock continues to allow thread safe access to process data structures such as the process control block (PCB). However, access to process address space and any structures currently not explicitly synchronized with spin locks are no longer guaranteed exclusive access solely by access mode. In the multithreaded environment, a new process level synchronization mechanism is required.

Because spin locks operate on a systemwide level and do not offer the process level granularity required for inner mode access synchronization in a multithreaded environment, a process level semaphore is necessary to serialize inner mode (kernel and executive) access. User and supervisor mode threads are allowed to run without any required synchronization.

The process level semaphore for inner mode synchronization is the inner mode (IM) semaphore. The IM semaphore is created in the first floating-point registers and execution data block (FRED) page in the balance set slot process for each process. In a multithreaded environment, a thread requiring inner mode access must acquire ownership of the IM semaphore. That is, two threads associated with the same process cannot execute in inner mode simultaneously. If the semaphore is owned by another thread, then the requesting thread spins until inner mode access becomes available, or until some specified timeout value has expired. ♦

14.5.3 Synchronization Using Process Priority

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

If you use process priority for synchronization, be aware that if the higher-priority process is blocked, either explicitly or implicitly (for example, when doing an I/O), the lower-priority process can run, resulting in corruption on the data of the higher process's activities.

Because each processor in a multiprocessor system, when idle, schedules its own work load, it is impossible to prevent all other processes in the system from executing. Moreover, because the scheduler guarantees only that the highest-priority and computable process is scheduled at any given time, it is impossible to prevent another process in an application from executing.

Thus, application programs that synchronize by process priority must be modified to use a different serialization method to run correctly in a multiprocessor system.

14.5.4 Synchronizing Multiprocess Applications

The operating system provides the following techniques to synchronize multiprocess applications:

- Common event flags
- Lock management system services
- Parallel processing (PPL$) run-time library procedures

The operating system provides basic event synchronization through event flags. Common event flags can be shared among cooperating processes running on a uniprocessor or in an SMP system, though the processes must be in the same user identification code (UIC) group. Thus, if you have developed an application that requires the concurrent execution of several processes, you can use event flags to establish communication among them and to synchronize their activity. A shared, or common, event flag can represent any event that is detectable and agreed upon by the cooperating processes. See Section 14.6 for information about using event flags.

The lock management system services—Enqueue Lock Request (SYS$ENQ), and Dequeue Lock Request (SYS$DEQ)—provide multiprocess synchronization tools that can be requested from all access modes. For details about using lock management system services, see Chapter 15.

The parallel processing run-time library procedures provide support for a number of different synchronization techniques suitable for user access-mode applications. These techniques include the following:

- Mutual exclusion implemented through an application-created semaphore or spin lock
- Event synchronization, in which one or more processes can wait for the occurrence of a user-defined event that is triggered by another process

- Barrier synchronization, in which multiple processes wait until a specified number of them have all reached a designated point in their execution

Section 14.7 describes the various PPL$ routines. The *OpenVMS RTL Parallel Processing (PPL$) Manual* provides more information.

Synchronization of access to shared data by a multiprocess application should be designed to support processes that execute concurrently on different members of an SMP system. Applications that share a global section can use VAX MACRO interlocked instructions or the equivalent in other languages to synchronize access to data in the global section. These applications can also use the lock management system services for synchronization.

14.5.5 Writing Applications for an Operating System Running in a Multiprocessor Environment

Most application programs that run on an operating system in a uniprocessor system also run without modification in a multiprocessor system. However, applications that access writable global sections or that rely on process priority for synchronizing tasks should be reexamined and modified according to the information contained in this section.

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

14.5.6 Synchronization Using Spin Locks

A **spin lock** is a device used by a processor to synchronize access to data that is shared by members of a symmetric multiprocessing (SMP) system. A spin lock enables a set of processors to serialize their access to shared data. The basic form of a spin lock is a bit that indicates the state of a particular set of shared data. When the bit is set, it shows that a processor is accessing the data. A bit is either tested and set or tested and cleared; it is atomic with respect to other threads of execution on the same or other processors.

A processor that needs access to some shared data tests and sets the spin lock associated with that data. To test and set the spin lock, the processor uses an interlocked bit-test-and-set instruction. If the bit is clear, the processor can have access to the data. This is called locking or acquiring the spin lock. If the bit is set, the processor must wait because another processor is already accessing the data.

Essentially, a waiting processor spins in a tight loop; it executes repeated bit test instructions to test the state of the spin lock. The term spin lock derives from this spinning. When the spin lock is in a loop, repeatedly testing the state of the spin lock, the spin lock is said to be in a state of busy wait. The busy wait ends when the processor accessing the data clears the bit with an interlocked operation to indicate that it is done. When the bit is cleared, the spin lock is said to be unlocked or released.

Spin locks are used by the operating system executive, along with the interrupt priority level (IPL), to control access to system data structures in a multiprocessor system.

See Section 14.7 for descriptions of how to use spin locks in your applications.

14.5.7 Writable Global Sections

A writable global section is an area of memory that can be accessed (read and modified) by more than one process. On uniprocessor or SMP systems, access to a single global section with an appropriate read or write instruction is atomic on VAX and Alpha systems. Therefore, no other synchronization is required.

An appropriate read or write on VAX systems is an instruction that is a naturally aligned byte, word, or longword, such as a MOV*x* instruction, where *x* is a B for a byte, W for a word, or L for a longword. On Alpha systems, an appropriate read or write instruction is a naturally aligned longword or quadword, for instance, an LD*x* or write ST*x* instruction where *x* is an L for an aligned longword or Q for an aligned quadword.

On both VAX and Alpha multiprocessor systems, for a read-modify-write sequence on a multiprocessor system, two or more processes can execute concurrently, one on each processor. As a result, it is possible that concurrently executing processes can access the same locations simultaneously in a writable global section. If this happens, only partial updates may occur, or data could be corrupted or lost, because the operation is not atomic. Unless proper interlocked instructions are used on VAX systems or load-locked/store-conditional instructions are used on Alpha systems, invalid data may result. You must use interlocked or load-locked /store-conditional instructions or other synchronizing techniques, such as locks or event flags.

On a uniprocessor or SMP system, access to multiple locations within a global section with read or write instructions or a read-modify-write sequence is not atomic on VAX and Alpha systems. On a uniprocessor system, an interrupt can occur that causes process preemption, allowing another process to run and access the data before the first process completes its work. On a multiprocessor system, two processes can access the global section simultaneously on different processors. You must use a synchronization technique such as a spin lock or event flags to avoid these problems.

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

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

14.6 Using Event Flags

Event flags are maintained by the operating system for general programming use in coordinating thread execution with asynchronous events. Programs can use event flags to perform a variety of signaling functions. Event flag services clear, set, and read event flags. They also place a thread in a wait state pending the setting of an event flag or flags.

Table 14–1 shows the two usage styles of event flags:

Table 14–1 Usage Styles of Event Flags

Style	Meaning
Explicit	Uses SET, CLEAR, and READ functions that are commonly used when one thread deals with multiple asynchronous events.
Implicit	Uses the SYS$SYNCH and wait form of system services when one or more threads wish to wait for a particular event. For multithreaded applications, only the implicit use of event flags is recommended.

The wait form of system services is a variant of asynchronous services; there is a service request and then a wait for the completion of the request. Digital recommends that an I/O status block (IOSB) be used with WAIT services. The IOSB prevents the service from completing prematurely, and also provides status information.

14.6.1 General Guidelines for Using Event Flags

Explicit use of event flags follows these general steps:

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

Implicit use of event flags may involve only step 4, or steps 1, 4, and 5.

Use run-time library routines and system services to accomplish these event flag tasks. Table 14–2 summarizes the event flag routines and services.

Table 14–2 Event Flag Routines and Services

Routine or Service	Task
LIB$FREE_EF	Deallocate a local event flag
LIB$GET_EF	Allocate any local event flag
LIB$RESERVE_EF	Allocate a specific local event flag
SYS$ASCEFC	Associate a common event flag cluster
SYS$CLREF	Clear a local or common event flag
SYS$DACEFC	Dissociate a common event flag cluster
SYS$DLCEFC	Delete common event flag cluster
SYS$READEF	Read a local or common event flag
SYS$SETEF	Set a local or common event flag
SYS$SYNCH	Wait for a local or common event flag to be set and for nonzero I/O status block—recommended to be used with threads
SYS$WAITFR	Wait for a specific local or common event flag to be set—not recommended to be used with threads

(continued on next page)

Table 14–2 (Cont.) Event Flag Routines and Services

Routine or Service	Task
SYS$WFLAND	Wait for several local or common event flags to be set—logical AND of event flags
SYS$WFLOR	Wait for one of several local or common event flags to be set—logical OR of event flags

Some system services set an event flag to indicate the completion or the occurrence of an event; the calling program can test the flag. Other system services use event flags to signal events to the calling process, such as SYS$ENQ(W), SYS$QIO(W), or SYS$SETIMR.

14.6.2 Introducing Local and Common Event Flag Numbers and Event Flag Clusters

Each event flag has a unique number; event flag arguments in system service calls refer to these numbers. For example, if you specify event flag 1 in a call to the SYS$QIO system service, then event flag 1 is set when the I/O operation completes.

To allow manipulation of groups of event flags, the flags are ordered in clusters of 32 numbers corresponding to bits 0 through 31 (<31:0>) in a longword. The clusters are also numbered from 0 to 4. The range of event flag numbers encompasses the flags in all clusters: event flag 0 is the first flag in cluster 0, event flag 32 is the first flag in cluster 1, and so on.

Event flags are divided into five clusters: two for local event flags and two for common event flags. There is also a special local cluster 4 that supports EFN 128.

- A local event flag cluster is process specific and is used to synchronize events within a process.
- A common event flag cluster can be shared by cooperating processes' UIC. A common event flag cluster is identified by name and is specific to a UIC group and VMScluster node. Before a process can use a common event flag cluster, it must explicitly "associate" with the cluster. (Association is described in Section 14.6.5.) Use them to synchronize events among images executing in different processes.
- A special local cluster 4 supports only EFN 128, symbolically EFN$C_ENF. EFN$C_ENF is intended for use with wait form services, such as SYS$QIOW and SYS$ENQW, or SYS$SYNCH system service. There is no need to reserve or free this event flag. Multiple threads of execution may concurrently use EFN$C_ENF without interference. With explicit event flag services, EFN$C_ENF behaves as if always set.

Table 14–3 summarizes the ranges of event flag numbers and the clusters to which they belong.

The same system services manipulate flags in either local and common event flag clusters. Because the event flag number implies the cluster number, you need not specify the cluster number when you call a system service that refers to an event flag.

When a system service requires an event flag cluster number as an argument, you need only specify the number of any event flag in the cluster. Thus, to read the event flags in cluster 1, you could specify any number in the range 32 through 63.

Table 14–3 Event Flags

Cluster Number	Flag Number	Type	Usage
0	0	Local	Default flag used by system routines.
0	1 to 23	Local	May be used in system routines. When an event flag is requested, it is not returned unless it has been previously and specifically freed by calls to LIB$FREE_EF.
0	24 to 31	Local	Reserved for Digital use only.
1	32 to 63	Local	Available for general use.
2	64 to 95	Common	Available for general use.
3	96 to 127	Common	Available for general use.
4	128	Local	Available for general use without explicit allocation.

14.6.3 Using EFN$C_ENF Local Event Flag

It is intended that EFN$C_ENF be used with the wait form of system services, or with SYS$SYNCH system service. EFN$C_ENF does not need to be initialized, nor does it need to be reserved or freed. Multiple threads of execution may concurrently use EFN$C_ENF without interference. When EFN$C_ENF is used with explicit event flag system services, it performs as if always set. It is recommended that ERN$C_ENF be used to eliminate the chance for event flag overlap.

14.6.4 Using Local Event Flags

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

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

1. To ensure that the event flag you are using is not accessed and changed by other users, allocate and deallocate local event flags. The *OpenVMS RTL Library (LIB$) Manual* describes routines you can use to allocate an arbitrary event flag (LIB$GET_EF), to allocate a particular event flag (LIB$RESERVE_EF), and to deallocate an event flag (LIB$FREE_EF) from the processwide pool of available local event flags. Similar routines do not exist for common event flags. If free, these routines return an event flag number.

   The LIB$GET_EF routine by default allocates flags from event flag cluster 1 (event flags 32 through 63). Event flags 1 through 32 (in event flag cluster 0) can also optionally be allocated by calls to LIB$GET_EF. To maintain compatibility with older application software that used event flags 1 through 23 in an uncoordinated fashion, these event flags must be initially marked as free by application calls to the LIB$FREE_EF routine before these flags can be allocated by subsequent calls to the LIB$GET_EF routine.

2. Before using the event flag, initialize it using the SYS$CLREF system service, unless you pass the event flag to a routine that clears it for you.
3. When an event that is relevant to other program components is completed, set the event flag with the SYS$SETEF system service.
4. A program component can read the event flag to determine what has happened and act accordingly. Use the SYS$READEF system service to read the event flag.
5. The program components that evaluate event flag status can be placed in a wait state. Then, when the event flag is set, execution is resumed. Use the SYS$WAITFR, SYS$WFLOR, SYS$WFLAND, or SYS$SYNCH routine to accomplish this task.
6. When the event flag is no longer required, free it by using the LIB$FREE_EF routine.

The following Fortran example uses LIB$GET_EF to choose a local event flag and then uses SYS$CLREF to set the event flag to 0 (clear the event flag). (Note that run-time library routines require an event flag number to be passed by reference, and system services require an event flag number to be passed by value.)

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

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

14.6.4.1 Example of Event Flag Services

Local event flags are used most commonly in conjunction with other system services. For example, you can use the Set Timer (SYS$SETIMR) system service to request that an event flag be set at a specific time of day or after a specific interval of time has passed. If you want to place a process in a wait state for a specified period of time, specify an event flag number for the SYS$SETIMR service and then use the Wait for Single Event Flag (SYS$WAITFR) system service, as shown in the C example that follows:

```
   .
   .
   .
main() {

        unsigned int status, daytim[1], efn=3;

/* Set the timer */
        status = SYS$SETIMR( efn,  /* efn - event flag */
                   &daytim,        /* daytim - expiration time */
                   0,              /* astadr - AST routine */
                   0,              /* reqidt - timer request id */
                   0);             /* flags */
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
/* Wait until timer expires */
        status = SYS$WAITFR( efn );
        if ((status & 1) != 1)
                LIB$SIGNAL( status );
   .
   .
   .
}
```

In this example, the **daytim** argument refers to a 64-bit time value. For details about how to obtain a time value in the proper format for input to this service, see Chapter 5.

14.6.5 Using Common Event Flags

Common event flags are manipulated like local event flags. However, before a process can use event flags in a common event flag cluster, the cluster must be created. The Associate Common Event Flag Cluster (SYS$ASCEFC) system service creates a common event flag cluster. When this common event flag cluster is created, it must be identified by a name string. (Section 14.6.5.1 explains the format of this string.) By calling SYS$ASCEFC, other processes in the same group can establish their association with the cluster so they can access flags in it. Each process that associates with the cluster must use the same name to refer to it; the SYS$ASCEFC system service establishes correspondence between the cluster name, and the cluster number that a process assigns to the cluster.

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

Different processes may associate the same name with different common event flag numbers; as long as the name and UIC group, or node are 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 identify the flag.

Common event flags act as a communication mechanism between images executing in different processes in the same group on the same OpenVMS cluster node. Common event flags are often used as a synchronization tool for other, more complicated communication techniques, such as logical names and global sections. For more information about using event flags to synchronize communication between processes, see Chapter 2.

If every cooperating process that is going to use a common event flag cluster has the necessary privilege or quota to create a cluster, the first process to call the SYS$ASCEFC system service creates the cluster.

The following example shows how a process might create a common event flag cluster named COMMON_CLUSTER and assign it a cluster number of 2:

```
   .
   .
   .
#include <descrip.h>
   .
   .
   .
        unsigned int status, efn=65;
        $DESCRIPTOR(name,"COMMON_CLUSTER"); /* Cluster name */
   .
   .
   .
/* Create cluster 2 */
        status = SYS$ASCEFC( efn, &name, 0, 0);
```

Other processes in the same group can now associate with this cluster. Those processes must use the same character string name to refer to the cluster; however, the cluster numbers they assign do not have to be the same.

14.6.5.1 Using the name Argument with SYS$ASCEFC

The **name** argument to the Associate Common Event Flag Cluster (SYS$ASCEFC) system service identifies the cluster that the process is creating or associating with. The **name** argument specifies a descriptor pointing to a character string.

Translation of the **name** argument proceeds in the following manner:

1. CEF$ is prefixed to the current name string, and the result is subjected to logical name translation.
2. If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the SYSGEN parameter LNM$C_MAXDEPTH.
3. The CEF$ prefix is stripped from the current name string that could not be translated. This current string is the cluster name.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE CEF$CLUS_RT CLUS_RT_001
```

Assume also that your program contains the following statements:

```
#include <ssdef.h>
#include <descrip.h>
   .
   .
   .
        unsigned int status;
        $DESCRIPTOR(name,"CLUS_RT"); /* Logical name of cluster */
   .
   .
   .
        status = SYS$ASCEFC( . . .  ,&name, . . .  );
```

The following logical name translation takes place:

1. CEF$ is prefixed to CLUS_RT.

2. CEF$CLUS_RT is translated to CLUS_RT_001. (Further translation is unsuccessful. When logical name translation fails, the string is passed to the service.)

There are two exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), the operating system strips the underscore and considers the resultant string to be the actual name (that is, further translation is not performed).
- If the name string is the result of a logical name translation, the name string is checked to see whether it has the **terminal** attribute. If it does, the operating system considers the resultant string to be the actual name (that is, further translation is not performed).

14.6.5.2 Temporary Common Event Flag Clusters

Common event flag clusters are either temporary or permanent. The **perm** argument to the SYS$ASCEFC system service defines whether the cluster is temporary or permanent.

Temporary clusters require an element of the creating process's quota for timer queue entries (TQELM quota). They are deleted when all processes associated with the cluster have disassociated. When the last image associated with a cluster is dissociated, the common event flag cluster is deleted. Clusters that are deleted after all images are dissociated are called temporary clusters. Dissociation can be performed explicitly with the Disassociate Common Event Flag Cluster (SYS$DACEFC) system service, or implicitly when the image that called SYS$ASCEFC exits.

14.6.5.3 Permanent Common Event Flag Clusters

If you have the PRMCEB privilege, you can create a permanent common event flag cluster (set the **perm** argument to 1 when you invoke SYS$ASCEFC). A permanent event flag cluster continues to exist until it is marked explicitly for deletion with the Delete Common Event Flag Cluster (SYS$DLCEFC) system service (requires the PRMCEB privilege). Once a permanent cluster is marked for deletion, it is like a temporary cluster; when the last image associated with the cluster is dissociated, the cluster is deleted.

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

Example 1

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

Example 2

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

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

Example 1

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

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

Example 2

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

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

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

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

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

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

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

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

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

14.6.6 Wait Form Services and SYS$SYNCH

Wait Form system services are a variant of asynchronous services in which there is a service request and then a wait for the completion of the request. The SYS$SYNCH system service checks the completion status of a system service that completes asynchronously. The service whose completion status is to be checked must have been called with the **efn** and **iosb** arguments specified. The SYS$SYNCH service uses the event flag and I/O status block of the service to be checked.

Digital recommends that only EFN$C_ENF be used for concurrent use of event flags.

14.6.7 Event Flag Waits

The following three system services place the process or thread in a wait state until an event flag or a group of event flags is set:

- The Wait for Single Event Flag (SYS$WAITFR) system service places the process or thread in a wait state until a *single* flag has been set.
- The Wait for Logical OR of Event Flags (SYS$WFLOR) system service places the process or thread in a wait state until *any one* of a specified group of event flags has been set.
- The Wait for Logical AND of Event Flags (SYS$WFLAND) system service places the process or thread in a wait state until *all* of a specified group of event flags have been set.

Another system service that accepts an event flag number as an argument is the Queue I/O Request (SYS$QIO) system service. The following example shows a program segment that issues two SYS$QIO system service calls and uses the SYS$WFLAND system service to wait until both I/O operations complete before it continues execution:

```
         .
         .
         .
              unsigned int status, efn1=1, efn2=2, mask;
         .
         .
         .
/* Issue first I/O request and check for error */
              status = SYS$QIO( efn1, . . .  )
              if ((status & 1) != 1)                                 1
                      LIB$SIGNAL(status);

/* Issue second I/O request and check for error */
              status = SYS$QIO( efn2, . . .  )
              if ((status & 1) != 1)
                      LIB$SIGNAL(status);

/* Wait until both complete and check for error */                     2
              mask = efn1 || efn2;
              status = SYS$WFLAND( efn1, mask );                       3
              if ((status & 1) != 1)
                      LIB$SIGNAL(status);
         .
         .
         .
```

1 The event flag argument is specified in each SYS$QIO request. Both of these event flags are in cluster 0.

2 After both I/O requests are queued successfully, the program calls the SYS$WFLAND system service to wait until the I/O operations complete. In this service call, the **efn** argument can specify any event flag number in the cluster containing the event flags to be waited for. The **mask** argument specifies to wait for flags 1 and 2.

3 Note that the SYS$WFLAND system service (and the other wait system services) waits for the event flag to be set; it does not wait for the I/O operation to complete. If some other event were to set the required event flags, the wait for event flag would complete prematurely. Use of event flags must be coordinated carefully.

14.6.8 Setting and Clearing Event Flags

System services that use event flags clear the event flag specified in the system service call before they queue the timer or I/O request. This ensures that the process knows the state of the event flag. If you are using event flags in local clusters for other purposes, be sure the flag's initial value is what you want before you use it.

The Set Event Flag (SYS$SETEF) and Clear Event Flag (SYS$CLREF) system services set and clear specific event flags. For example, the following system service call clears event flag 32:

```
$CLREF_S EFN=#32
```

The SYS$SETEF and SYS$CLREF services return successful status codes that indicate whether the specified flag was set or cleared when the service was called. The caller can thus determine the previous state of the flag, if necessary. The codes returned are SS$_WASSET and SS$_WASCLR.

All event flags in a common event flag cluster are initially clear when the cluster is created. Section 14.6.9 describes the creation of common event flag clusters.

14.6.9 Example of Using a Common Event Flag Cluster

The following example shows four cooperating processes that share a common event flag cluster. The processes named ORION, CYGNUS, LYRA, and PEGASUS are in the same group.

```
/* **** Process ORION **** */

   .
   .
   .
        unsigned int status, efn=64;
        $DESCRIPTOR(name,"TITUS");
   .
   .
   .
/* Create a common cluster */
                                                              1
        status = SYS$ASCEFC(efn, &name, . . .  );          2
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        mask = efn1 || efn2 || efn3;

/* Wait for flags 1, 2, and 3 */

        status = SYS$WFLAND(efn, mask);                       3
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
/* Dissociated cluster */

        status = SYS$DACEFC(efn);                             4

/* **** Process CYGNUS **** */

   .
   .
   .
        unsigned int status, efn1=64,efn2=64;
        $DESCRIPTOR(name,"TITUS");
   .
   .
   .
        status = SYS$ASCEFC(efn1,&name, . . .  );          5
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

/* Set event flag 1 and check for error */
        status = SYS$SETEF(efn2);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        status = SYS$DACEFC(efn1);

/* **** Process LYRA **** */
        $DESCRIPTOR(name,"TITUS");
 . . .
```

```
/* Associate with cluster 3 */
        status = SYS$ASCEFC(efn, &name);                          6
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

/* Set event flag 3 and check for error */
        status = SYS$SETEF(efn2);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        status = SYS$DACEFC(efn1);

/* **** Process PEGASUS **** */

/* Associate with cluster  */
        status = SYS$ASCEFC(efn, &name);                            7
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

/* Wait for flag 1 and check for error */
        status = SYS$WAITFR(efn2);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);

/* Set event flag 2 and check for error */
        status = SYS$SETEF(efn2);
        if ((status & 1) != 1)
                LIB$SIGNAL(status);
   .
   .
   .
        status = SYS$DACEFC(efn1);
```

1 Assume that ORION is the first process to issue the SYS$ASCEFC system service and, therefore, is the creator of the cluster. Because this is a newly created cluster, all event flags in it are clear.

2 The argument **name** in the SYS$ASCEFC system service call is a pointer to the descriptor CNAME for the name to be assigned to the cluster; in this example, the cluster is named TITUS. This service call associates this name with cluster 2 of process ORION and contains event flags 64 through 95. Cooperating processes CYGNUS, LYRA, and PEGASUS must use the same character string name to refer to this cluster.

3 The continuation of process ORION depends on work done by processes CYGNUS, LYRA, and PEGASUS. The SYS$WFLAND system service call specifies a mask indicating the event flags that must be set before process ORION can continue. The mask in this example (^B1110) indicates that the second, third, and fourth flags in the cluster must be set.

4 When all three event flags are set, process ORION continues execution and calls the SYS$DACEFC system service. Because ORION did not specify the **perm** argument when it created the cluster, TITUS is deleted.

5 Process CYGNUS executes, associates with the cluster, sets event flag 65 (flag 1 in the cluster), and dissociates.

6 Process LYRA associates with the cluster, but instead of referring to it as cluster 2, it refers to it as cluster 3 (with event flags in the range 96 through 127). Thus, when process LYRA sets flag 99, it sets flag 3 in TITUS.

7 Process PEGASUS associates with the cluster, waits for an event flag set by process CYGNUS, and sets an event flag itself.

14.6.10 Example of Using Event Flag Routines and Services

This section contains an example of how to use event flag services.

Common event flags are often used for communicating between a parent process and a created subprocess. In the following example, REPORT.FOR creates a subprocess to execute REPORTSUB.FOR, which performs a number of operations.

After REPORTSUB.FOR performs its first operation, the two processes can perform in parallel. REPORT.FOR and REPORTSUB.FOR use the common event flag cluster named JESSIER to communicate.

REPORT.FOR associates the cluster name with a common event flag cluster, creates a subprocess to execute REPORTSUB.FOR and then waits for REPORTSUB.FOR to set the first event flag in the cluster. REPORTSUB.FOR performs its first operation, associates the cluster name JESSIER with a common event flag cluster, and sets the first flag. From then on, the processes execute concurrently.

```
                         REPORT.FOR
.
.
.
! Associate common event flag cluster
STATUS = SYS$ASCEFC (%VAL(64),
2                    'JESSIER',,)
IF (.NOT. STATUS) CALL LIB$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$WAITFR (%VAL(64))
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
.
.
.

REPORTSUB.FOR
.
.
.
! Do operations necessary for
! continuation of parent process.
.
.
.
! Associate common event flag cluster
STATUS = SYS$ASCEFC (%VAL(64),
2                    'JESSIER',,)
IF (.NOT. STATUS)
2  CALL LIB$SIGNAL (%VAL(STATUS))
```

```
! Set flag for parent process to resume
STATUS = SYS$SETEF (%VAL(64))
.
.
.
```

14.7 Synchronizing Operations with Parallel Processing Run-Time Routines

The Parallel Processing facility (PPL$) consists of routines for synchronizing program processing in a synchronous multiprocessing configuration. The routines provide for the following capabilities:

- Creating subprocesses
- Synchronizing program execution using spin locks
- Synchronizing program execution using semaphores
- Synchronizing program execution using barriers
- Setting up global sections of memory for shared use

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

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

14.7.1 Using Subprocesses

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

To delete one or more subprocesses created with PPL$SPAWN, use PPL$STOP.

14.7.2 Using Spin Locks

To ensure that only one process at a time can access a critical region or physical resource of a parallel task, you can use spin locks. A spin lock is a lock on a critical region where the lock constantly tests to determine whether access to that region is available. Because of the constant testing, this technique is CPU intensive. An alternative technique to ensure single access is to use semaphores. Refer to Section 14.7.3 for more information on using semaphores.

The three spinlock routines are as follows:

- PPL$CREATE_SPIN_LOCK—Creates and initializes a simple lock. An identifier is returned for subsequent reference to this spin lock.
- PPL$SEIZE_SPIN_LOCK—Acquires a spin lock that has been created with PPL$CREATE_SPIN_LOCK. Use the identifier returned by PPL$CREATE_SPIN_LOCK to refer to the lock you want to acquire.

- PPL$RELEASE_SPIN_LOCK—Releases a spin lock. Use the identifier returned by PPL$CREATE_SPIN_LOCK to refer to the lock you want to release. Once this routine has freed the lock, another process can acquire the lock.

14.7.3 Using Semaphores

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

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

The semaphore routines are as follows:

- PPL$CREATE_SEMAPHORE—Creates and initializes a semaphore and creates a waiting queue that keeps track of processes waiting for the semaphore.
- PPL$DECREMENT_SEMAPHORE—Decrements the value of a semaphore. If the value of the semaphore is 0, the process requesting the semaphore can be placed in a wait state until the semaphore value increases.
- PPL$INCREMENT_SEMAPHORE—Increments the value of a semaphore to indicate that the resource can be accessed. If a process is waiting for the semaphore, PPL$INCREMENT_SEMAPHORE wakes up the process and removes it from the wait queue.
- PPL$READ_SEMAPHORE—Returns the value of the requested semaphore.

14.7.4 Using Barrier Synchronization

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

The barrier routines are as follows:

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

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

14.8 Synchronizing Operations with Synchronous and Asynchronous System Services

A number of system services can be executed either synchronously or asynchronously such as the following:

- SYS$GETJPI and SYS$GETJPIW
- SYS$QIO and SYS$QIOW

The W at the end of the system service name indicates the synchronous version of the service.

The asynchronous version of a system service queues a request and immediately returns control to your program pending the completion of the request. You can perform other operations while the system service executes. To avoid data corruptions, you should not attempt any read or write access to any of the buffers or itemlists referenced by the system service call prior to the completion of the asynchronous portion of the system service call. Further, no self-referential or self-modifying itemlists should be used.

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

The following example show the use of the SYS$GETJPI system service:

```
! 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 as if you had used the asynchronous version followed immediately by a call to SYS$SYNCH; however, it behaves this way only if you specify a status block. If you omit the I/O status block, the result is as though you called the asynchronous version followed by a call to SYS$WAITFR. 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.

15

Synchronizing Access to Resources

This chapter describes the use of the lock manager to synchronize access to shared resources. It contains the following sections:

Section 15.1 describes how the lock manager synchronizes processes to a specified resource.

Section 15.2 describes the concepts of resources and locks.

Section 15.3 describes how to use the SYS$ENQ and SYS$ENQW system services to queue lock requests.

Section 15.4 describes specialized features of locking techniques.

Section 15.5 describes how to use the SYS$DEQ system service to dequeue the lock.

Section 15.6 describes how applications can perform local buffer caching.

Section 15.7 presents a code example of how to use lock management services.

15.1 Synchronizing Operations with the Lock Manager

Cooperating processes can use the lock manager to synchronize access to a shared resource (for example, a file, program, or device). This synchronization is accomplished by allowing processes to establish locks on named resources. All processes that access the shared resources must use the lock management services; otherwise, the resources are not effective.

Note

The use of the term *resource* throughout this chapter means shared resource.

To synchronize access to resources, the lock management services provide a mechanism that allows processes to wait in a queue until a particular resource is available.

The lock manager does not ensure proper access to the resource; rather, the programs must respect the rules for using the lock manager. The rules required for proper synchronization to the resource are as follows:

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

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

Table 15–1 summarizes the lock manager services.

Table 15–1 Lock Manager Services

Routine	Description
SYS$ENQ(W)	Queues a new lock or lock conversion on a resource
SYS$DEQ	Releases locks and cancel lock requests
SYS$GETLKI(W)	Gets information about the lock database

15.2 Concepts of Resources and Locks

A resource can be any entity on the operating system (for example, files, data structures, databases, executable routines). When two or more processes access the same resource, you often need to control their access to the resource. You do not want to have one process reading the resource while another process writes new data, because a writer can quickly invalidate anything being read by a reader. The lock management system services allow processes to associate a name with a resource and request access to that resource. Lock modes enable processes to indicate how they want to share access with other processes.

To use the lock management system services, a process must request access to a resource (request a lock) using the Enqueue Lock Request (SYS$ENQ) system service. Three arguments are required to the SYS$ENQ system service for new locks:

- Resource name—The lock management services use the resource name to look for other lock requests that use the same name.
- Lock mode to be associated with the requested lock—The lock mode indicates how the process wants to share the resource with other processes.
- Address of a lock status block—The lock status block receives the completion status for a lock request and the lock identification. The lock identification is used to refer to a lock request after it has been queued.

The lock management services compare the lock mode of the newly requested lock to the mode of other locks with the same resource name. New locks are granted in the following instances:

- If no other process has a lock on the resource.
- If another process has a lock on the resource and the mode of the new request is compatible with the existing lock.

- If another process already has a lock on the resource and the mode of the new request is not compatible with the lock mode of the existing lock, the new request is placed in a queue, where it waits until the resource becomes available. When the resource becomes available, the process is notified that the lock has been granted.

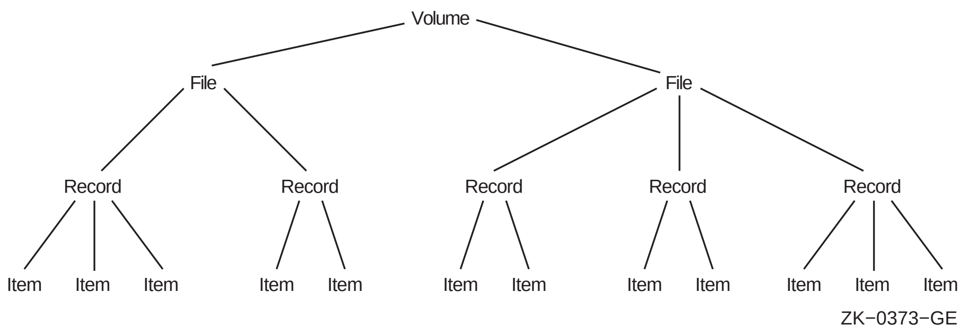
Processes can also use the SYS$ENQ system service to change the lock mode of a lock. This is called a **lock conversion.**

15.2.1 Resource Granularity

Many resources can be divided into smaller parts. As long as a part of a resource can be identified by a resource name, the part can be locked. The term **resource granularity** describes the part of the resource being locked.

Figure 15–1 depicts a model of a database. The database is divided into areas, such as a file, which in turn are subdivided into records. The records are further divided into items.

Figure 15–1 Model Database



The processes that request locks on the database shown in Figure 15–1 may lock the whole database, an area in the database, a record, or a single item. Locking the entire database is considered locking at a **coarse granularity**; locking a single item is considered locking at a **fine granularity**.

In this example, overall access to the database can be represented by a root resource name. Access to areas in the database or records within areas can be represented by sublocks.

Root resources consist of the following:

- Resource domain
- Resource name
- Access mode

Subresources consist of the following:

- Parent resource
- Resource name
- Access mode

15.2.2 Resource Domains

Because resource names are arbitrary names chosen by applications, one application may interfere (either intentionally or unintentionally) with another application. Unintentional interference can be easily avoided by careful design, such as using a registered facility name as a prefix for all root resource names used by an application.

Intentional interference can be prevented by using **resource domains**. A resource domain is a namespace for root resource names and is identified by a number. Resource domain 0 is used as a system resource domain. Usually, other resource domains are used by the UIC group corresponding to the domain number.

By using the SYS$SET_RESOURCE_DOMAIN system service, a process can gain access to any resource domain subject to normal operating system access control. By default, each resource domain allows read, write, and lock access by members of the corresponding UIC group. See the *OpenVMS Guide to System Security* for more information about access control.

15.2.3 Resource Names

The lock management system services refer to each resource by a name composed of the following four parts:

- A name specified by the caller
- The caller's access mode
- The caller's UIC group number (unless the resource is systemwide)
- The identification of the lock's parent (optional)

For two resources to be considered the same, these four parts must be identical for each resource.

The name specified by the process represents the resource being locked. Other processes that need to access the resource must refer to it using the same name. The correlation between the name and the resource is a convention agreed upon by the cooperating processes.

The access mode is determined by the caller's access mode unless a less privileged mode is specified in the call to the SYS$ENQ system service. Access modes, their numeric values, and their symbolic names are discussed in the *OpenVMS Calling Standard*.

The default resource domain is selected by the UIC group number for the process. The system domain can be accessed by setting the LCK$M_SYSTEM when you request a new root lock. Other domains can be accessed using the optional RSDM_ID parameter to SYS$ENQ. You need the SYSLCK user privilege to request systemwide locks from user or supervisor mode. No additional privilege is required to request systemwide locks from executive or kernel mode.

When a lock request is queued, it can specify the identification of a parent lock, at which point it becomes a sublock (see Section 15.4.8). However, the parent lock must be granted, or the lock request is not accepted. This enables a process to lock a resource at different degrees of granularity.

15.2.4 Choosing a Lock Mode

The mode of a lock determines whether the resource can be shared with other lock requests. Table 15–2 describes the six lock modes.

Table 15–2 Lock Modes

Mode Name	Meaning
LCK$K_NLMODE	Null mode. This mode grants no access to the resource. The null mode is typically used as an indicator of interest in the resource or as a placeholder for future lock conversions.
LCK$K_CRMODE	Concurrent read. This mode grants read access to the resource and allows sharing of the resource with other readers. The concurrent read mode is generally used when additional locking is being performed at a finer granularity with sublocks or to read data from a resource in an "unprotected" fashion (allowing simultaneous writes to the resource).
LCK$K_CWMODE	Concurrent write. This mode grants write access to the resource and allows sharing of the resource with other writers. The concurrent write mode is typically used to perform additional locking at a finer granularity, or to write in an "unprotected" fashion.
LCK$K_PRMODE	Protected read. This mode grants read access to the resource and allows sharing of the resource with other readers. No writers are allowed access to the resource. This is the traditional "share lock."
LCK$K_PWMODE	Protected write. This mode grants write access to the resource and allows sharing of the resource with users at concurrent read mode. No other writers are allowed access to the resource. This is the traditional "update lock."
LCK$K_EXMODE	Exclusive. The exclusive mode grants write access to the resource and prevents the sharing of the resource with any other readers or writers. This is the traditional "exclusive lock."

15.2.5 Levels of Locking and Compatibility

Locks that allow the process to share a resource are called **low-level locks**; locks that allow the process almost exclusive access to a resource are called **high-level locks**. Null and concurrent read mode locks are considered low-level locks; protected write and exclusive mode locks are considered high-level. The lock modes, from lowest- to highest-level access, are:

- Null
- Concurrent read
- Concurrent write
  Protected read
- Protected write
- Exclusive

Note that the concurrent write and protected read modes are considered to be of equal level.

Locks that can be shared with other locks are said to have compatible lock modes. High-level lock modes are less compatible with other lock modes than are low-level lock modes. Table 15–3 shows the compatibility of the lock modes.

Table 15–3 Compatibility of Lock Modes

Mode of Requested Lock	Mode of Currently Granted Locks					
	NL	**CR**	**CW**	**PR**	**PW**	**EX**
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No
PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

Key to Lock Modes:

NL—Null
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive

15.2.6 Lock Management Queues

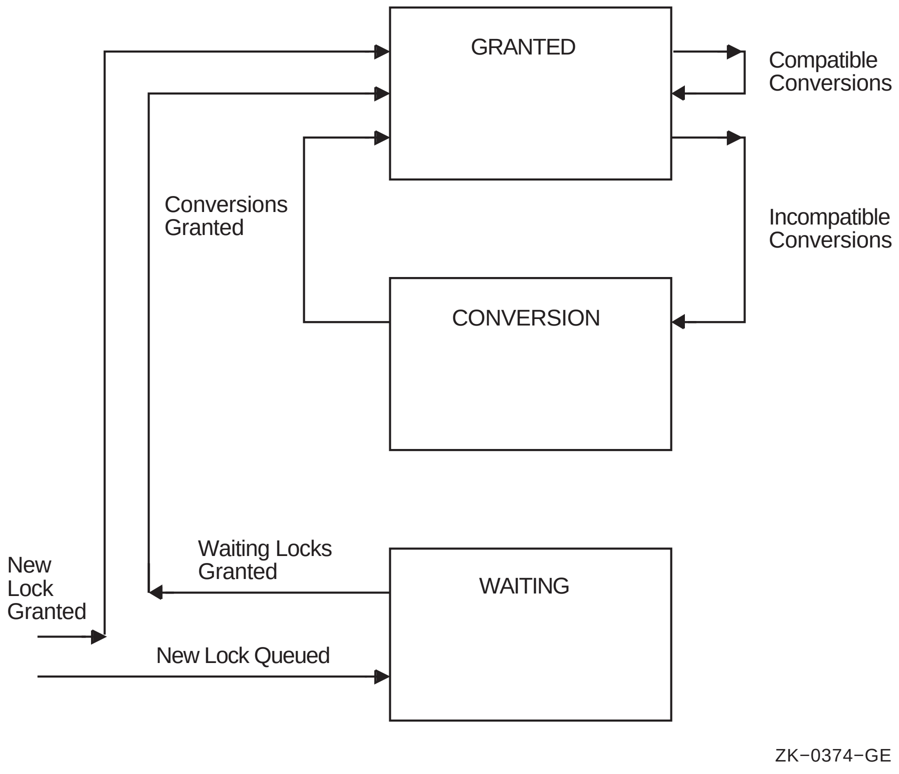
A lock on a resource can be in one of the following three states:

- Granted—The lock request has been granted.
- Waiting—The lock request is waiting to be granted.
- Conversion—The lock request has been granted at one mode and is waiting to be granted a high-level lock mode.

A queue is associated with each of the three states (see Figure 15–2).

When you request a new lock, the lock management services first determine whether the resource is currently known (that is, if any other processes have locks on that resource). If the resource is new (that is, if no other locks exist on the resource), the lock management services create an entry for the new resource and the requested lock. If the resource is already known, the lock management services determine whether any other locks are waiting in either the conversion or the waiting queue. If other locks are waiting in either queue, the new lock request is queued at the end of the waiting queue. If both the conversion and waiting queues are empty, the lock management services determine whether the new lock is compatible with the other granted locks. If the lock request is compatible, the lock is granted; if it is not compatible, it is placed in the waiting queue. You can use a flag bit to direct the lock management services not to queue a lock request if one cannot be granted immediately.

Figure 15–2 Three Lock Queues



15.2.7 Concepts of Lock Conversion

Lock conversions allow processes to change the level of locks. For example, a process can maintain a low-level lock on a resource until it limits access to the resource. The process can then request a lock conversion.

You specify lock conversions by using a flag bit (see Section 15.4.6) and a lock status block. The lock status block must contain the lock identification of the lock to be converted. If the new lock mode is compatible with the currently granted locks, the conversion request is granted immediately. If the new lock mode is incompatible with the existing locks in the granted queue, the request is placed in the conversion queue. The lock retains its old lock mode and does not receive its new lock mode until the request is granted.

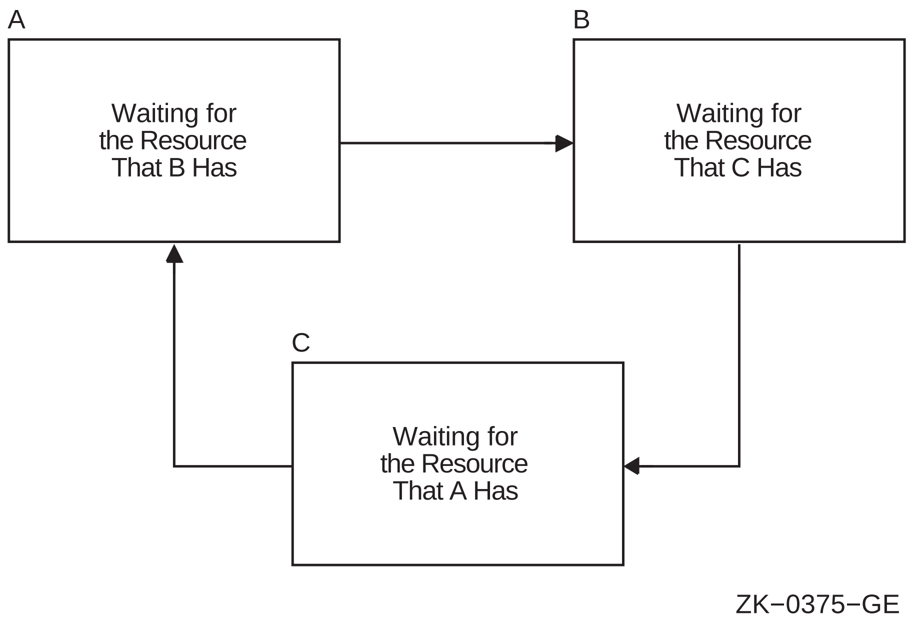
When a lock is dequeued or is converted to a lower-level lock mode, the lock management services inspect the first conversion request on the conversion queue. The conversion request is granted if it is compatible with the locks currently granted. Any compatible conversion requests immediately following are also granted. If the conversion queue is empty, the waiting queue is checked. The first lock request on the waiting queue is granted if it is compatible with the locks currently granted. Any compatible lock requests immediately following are also granted.

15.2.8 Deadlock Detection

A deadlock occurs when any group of locks are waiting for each other in a circular fashion.

In Figure 15–3, three processes have queued requests for resources that cannot be accessed until the current locks held are dequeued (or converted to a lower lock mode).

Figure 15–3 Deadlock


A
Waiting for the Resource That B Has
B
Waiting for the Resource That C Has
C
Waiting for the Resource That A Has
ZK−0375−GE

If the lock management services determine that a deadlock exists, the services choose a process to break the deadlock. The chosen process is termed the **victim.** If the victim has requested a new lock, the lock is not granted; if the victim has requested a lock conversion, the lock is returned to its old lock mode. In either case, the status code SS$_DEADLOCK is placed in the lock status block. Note that granted locks are never revoked; only waiting lock requests can receive the status code SS$_DEADLOCK.

Note

Programmers must not make assumptions regarding which process is to be chosen to break a deadlock.

15.3 Queuing Lock Requests

You use the SYS$ENQ or SYS$ENQW system service to queue lock requests. SYS$ENQ queues a lock request and returns; SYS$ENQW queues a lock request, waits until the lock is granted, and then returns. When you request new locks, the system service call must specify the lock mode, address of the lock status block, and resource name.

The format for SYS$ENQ and SYS$ENQW is as follows:

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

The following example illustrates a call to SYS$ENQW:

```
#include <stdio.h>
#include <descrip.h>
#include <lckdef.h>

/* Declare a lock status block */

struct lock_blk {
                  unsigned short  condition,reserved;
                  unsigned int lock_id;
}lksb;

   .
   .
   .
        unsigned int status, lkmode=LCK$K_PRMODE;
        $DESCRIPTOR(resource,"STRUCTURE_1");

/* Queue a request for protected read mode lock */
        status = SYS$ENQW(0,     /* efn - event flag */
                 lkmode,         /* lkmode - lock mode requested */
                 &lksb,          /* lksb - lock status block */
                 0,              /* flags */
                 &resource,      /* resnam - name of resource */
                 0,              /* parid - parent lock id */
                 0,              /* astadr - AST routine */
                 0,              /* astprm - AST parameter */
                 0,              /* blkast - blocking AST */
                 0,              /* acmode - access mode */
                 0);             /* rsdm_id - resource domain id */

}
```

In this example, a number of processes access the STRUCTURE_1 data structure. Some processes read the data structure; others write to the structure. Readers must be protected from reading the structure while it is being updated by writers. The reader in the example queues a request for a protected read mode lock. Protected read mode is compatible with itself, so all readers can read the structure at the same time. A writer to the structure uses protected write or exclusive mode locks. Because protected write mode and exclusive mode are not compatible with protected read mode, no writers can write the data structure until the readers have released their locks, and no readers can read the data structure until the writers have released their locks.

Table 15–3 shows the compatibility of lock modes.

15.3.1 Example of Requesting a Null Lock

The program segment in Example 15–1 requests a null lock for the resource named TERMINAL. After the lock is granted, the program requests that the lock be converted to an exclusive lock. Note that, after SYS$ENQW returns, the program checks the status of the system service and the status returned in the lock status block to ensure that the request completed successfully. (The lock mode symbols are defined in the $LCKDEF module of the system macro library.)

Example 15–1 Requesting a Null Lock

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

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

15.4 Advanced Locking Techniques

The previous sections discuss locking techniques and concepts that are useful to all applications. The following sections discuss specialized features of the lock manager.

15.4.1 Synchronizing Locks

The SYS$ENQ system service returns control to the calling program when the lock request is queued. The status code in R0 indicates whether the request was queued successfully. After the request is queued, the procedure cannot access the resource until the request is granted. A procedure can use three methods to check that a request has been granted:

- Specify the number of an event flag to be set when the request is granted, and wait for the event flag.
- Specify the address of an AST routine to be executed when the request is granted.
- Poll the lock status block for a return status code that indicates that the request has been granted.

These methods of synchronization are identical to the synchronization techniques used with the SYS$QIO system services (described in Chapter 9).

The $ENQW macro performs synchronization by combining the functions of the SYS$ENQ system service and the Synchronize (SYS$SYNCH) system service. The $ENQW macro has the same arguments as the $ENQ macro. It queues the lock request and then places the program in an event flag wait state (LEF) until the lock request is granted.

15.4.2 Notification of Synchronous Completion

The lock management services provide a mechanism that allows processes to determine whether a lock request is granted synchronously, that is, if the lock is not placed on the conversion or waiting queue. This feature can be used to improve performance in applications where most locks are granted synchronously (as is normally the case).

If the flag bit LCK$M_SYNCSTS is set and a lock is granted synchronously, the status code SS$_SYNCH is returned in R0; no event flag is set, and no AST is delivered.

If the request is not completed synchronously, the success code SS$_NORMAL is returned; event flags or AST routines are handled normally (that is, the event flag is set, and the AST is delivered when the lock is granted).

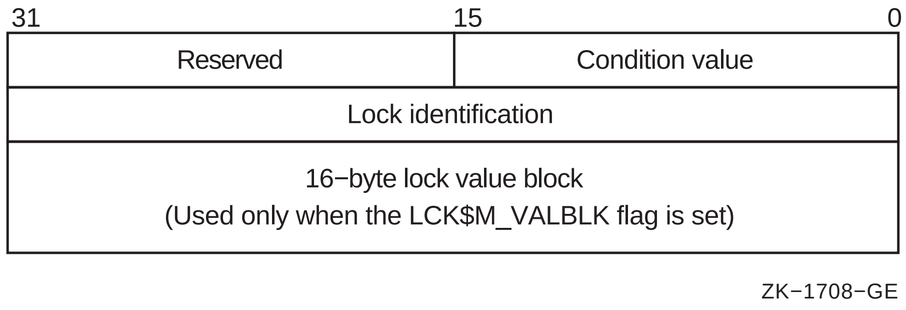
15.4.3 Expediting Lock Requests

A request can be expedited (granted immediately) if its requested mode, when granted, does not block any currently queued requests from being granted. The LCK$M_EXPEDITE flag is specified in the SYS$ENQ operation to expedite a request. Currently, only NLMODE requests can be expedited. A request to expedite any other lock mode fails with SS$_UNSUPPORTED status.

15.4.4 Lock Status Block

The lock status block receives the final completion status and the lock identification, and optionally contains a lock value block (see Figure 15–4). When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. For more information about the Dequeue Lock Request (SYS$DEQ) system service, see Section 15.5.

Figure 15–4 Lock Status Block



The status code is placed in the lock status block only when the lock is granted or when errors occur in granting the lock.

The uses of the lock value block are described in Section 15.6.1.

15.4.5 Blocking ASTs

In some applications that use the lock management services, a process must know whether it is preventing another process from locking a resource. The lock management services inform processes of this through the use of blocking ASTs. When the lock prevents another lock from being granted, the blocking routine is delivered as an AST to the process. Blocking ASTs are not delivered when the state of the lock is either Conversion or Waiting.

To enable blocking ASTs, the **blkast** argument of the SYS$ENQ system service must contain the address of a blocking AST service routine. The **astprm** argument is used to pass a parameter to the blocking AST. For more information about ASTs and AST service routines, see Chapter 4. Some uses of blocking ASTs are also described in that chapter.

15.4.6 Lock Conversions

Lock conversions perform three main functions:

- Maintaining a low-level lock and converting it to a higher lock mode when necessary
- Maintaining values stored in a resource lock value block (described in the following paragraphs)
- Improving performance in some applications

A procedure normally needs an exclusive (or protected write) mode lock while writing data. The procedure should not keep the resource exclusively locked all the time, however, because writing might not always be necessary. Maintaining an exclusive or protected write mode lock prevents other processes from accessing the resource. Lock conversions allow a process to request a low-level lock at first and convert the lock to a high-level lock mode (protected write mode, for example) only when it needs to write data.

Some applications of locks require the use of the lock value block. If a version number or other data is maintained in the lock value block, you need to maintain at least one lock on the resource so that the value block is not lost. In this case, processes convert their locks to null locks, rather than dequeuing them when they have finished accessing the resource.

In order to improve performance in some applications, all resources that might be locked are locked with null locks during initialization. You can convert the null locks to higher-level locks as needed. Usually a conversion request is faster than a new lock request because the necessary data structures have already been built. However, maintaining any lock for the life of a procedure uses system dynamic memory. Therefore, the approach of creating all necessary locks as null locks and converting them as needed improves performance at the expense of increased storage requirements.

Note

If you specify the flag bit LCK$M_NOQUEUE on a lock conversion and the conversion fails, the new blocking AST address and parameter specified in the conversion request replace the blocking AST address and parameter specified in the previous SYS$ENQ request.

Queuing Lock Conversions

To perform a lock conversion, a procedure calls the SYS$ENQ system service with the flag bit LCK$M_CONVERT. Lock conversions do not use the **resnam**, **parid**, **acmode**, or **prot** argument. The lock being converted is identified by the lock identification contained in the lock status block. The following program shows a simple lock conversion. Note that the lock must be granted before it can be converted.

```
#include <stdio.h>
#include <descrip.h>
#include <lckdef.h>

/* Declare a lock status block */

struct lock_blk {
                unsigned short lkstat, reserved;
                unsigned int lock_id;
}lksb;

   .
   .
   .
        unsigned int status, lkmode, flags;
        $DESCRIPTOR(resource,"STRUCTURE_1");
   .
   .
   .
        lkmode = LCK$K_NLMODE;

/* Queue a request for protected read mode lock */
        status = SYS$ENQW(0,    /* efn - event flag */
                  lkmode,       /* lkmode - lock mode */
                  &lksb,        /* lksb - lock status block */
                  0,            /* flags */
                   &resource,   /* resnam - name of resource */
                  0, 0, 0, 0, 0, 0);
   .
   .
   .
        lkmode = LCK$K_PWMODE;
        flags = LCK$M_CONVERT;

/* Queue a request for protected write mode lock */
        status = SYS$ENQW(0,     /* efn - event flag */
                  lkmode,        /* lkmode - lock mode */
                  &lksb,         /* lksb - lock status block */
                  flags,         /* flags */
                  0, 0, 0, 0, 0, 0, 0);
   .
   .
   .
}
```

15.4.7 Forced Queuing of Conversions

It is possible to force certain conversions to be queued that would otherwise be granted. A conversion request with the LCK$M_QUECVT flag set is forced to wait behind any already queued conversions.

The conversion request is granted immediately if no conversions are already queued.

The QUECVT behavior is valid only for a subset of all possible conversions. Table 15–4 defines the legal set of conversion requests for LCK$M_QUECVT. Illegal conversion requests fail with SS$_BADPARAM returned.

Table 15–4 Legal QUECVT Conversions

	Lock Mode to Which Lock Is Converted					
Lock Mode at Which Lock Is Held	**NL**	**CR**	**CW**	**PR**	**PW**	**EX**
NL	No	Yes	Yes	Yes	Yes	Yes
CR	No	No	Yes	Yes	Yes	Yes
CW	No	No	No	Yes	Yes	Yes
PR	No	No	Yes	No	Yes	Yes
PW	No	No	No	No	No	Yes
EX	No	No	No	No	No	No

Key to Lock Modes:

NL—Null
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive

15.4.8 Parent Locks

When a lock request is queued, you can declare a parent lock for the new lock. A lock that has a parent is called a **sublock.** To specify a parent lock, the lock identification of the parent lock is passed in the **parid** argument to the SYS$ENQ system service. A parent lock must be granted before the sublocks belonging to the parent can be granted.

The benefits of specifying parent locks are as follows:

- Low-level locks (concurrent read or concurrent write) can be held at a coarse granularity, such as files, whereas high-level (protected write or exclusive mode) sublocks are held on resources of a finer granularity, such as records or data items.
- Resources names are unique with each parent; parent locks are part of the resource name.

The following paragraphs describe the use of parent locks.

Assume that a number of processes need to access a database. The database can be locked at two levels: the file and individual records. For updating all the records in a file, locking the whole file and updating the records without additional locking is faster and more efficient. But for updating selected records, locking each record as it is needed is preferable.

To use parent locks in this way, all processes request locks on the file. Processes that need to update all records must request protected write or exclusive mode locks on the file. Processes that need to update individual records request concurrent write mode locks on the file and then use sublocks to lock the individual records in protected write or exclusive mode.

In this way, the processes that need to access all records can do so by locking the file, while processes that share the file can lock individual records. A number of processes can share the file-level lock at concurrent write mode while their sublocks update selected records.

VAX

On VAX systems, the number of levels of sublocks is limited by the size of the interrupt stack. If the limit is exceeded, the error status SS$_EXDEPTH is returned. The size of the interrupt stack is controlled by the SYSGEN parameter INTSTKPAGES. The default value for INTSTKPAGES allows 32 levels of sublocks. For more information about SYSGEN and INTSTKPAGES, see the *OpenVMS System Manager's Manual*. ♦

Alpha

On Alpha systems, the number of levels of sublocks is limited by the size of the kernel stack. If the limit is exceeded, the error status SS$_EXDEPTH is returned. The size of the kernel stack is controlled by the SYSGEN parameter KSTACKPAGES. ♦

15.4.9 Lock Value Blocks

The lock value block is an optional, 16-byte extension of a lock status block. The first time a process associates a lock value block with a particular resource, the lock management services create a resource lock value block for that resource. The lock management services maintain the resource lock value block until there are no more locks on the resource.

To associate a lock value block with a resource, the process must set the flag bit LCK$M_VALBLK in calls to the SYS$ENQ system service. The lock status block **lksb** argument must contain the address of the lock status block for the resource.

When a process sets the flag bit LCK$M_VALBLK in a lock request (or conversion request) and the request (or conversion) is granted, the contents of the resource lock value block are written to the lock value block of the process.

When a process sets the flag bit LCK$M_VALBLK on a conversion from protected write or exclusive mode to a lower mode, the contents of the process's lock value block are stored in the resource lock value block.

In this manner, processes can pass the value in the lock value block along with the ownership of a resource.

Table 15–5 shows how lock conversions affect the contents of the process's and the resource's lock value block.

Table 15–5 Effect of Lock Conversion on Lock Value Block

	Lock Mode to Which Lock Is Converted					
Lock Mode at Which Lock Is Held	**NL**	**CR**	**CW**	**PR**	**PW**	**EX**
NL	Return	Return	Return	Return	Return	Return
CR	Neither	Return	Return	Return	Return	Return
CW	Neither	Neither	Return	Return	Return	Return
PR	Neither	Neither	Neither	Return	Return	Return
PW	Write	Write	Write	Write	Write	Return
EX	Write	Write	Write	Write	Write	Write

Key to Lock Modes:

NL—Null
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive

Key to Effects:

Return—The contents of the resource lock value block are returned to the lock value block of the process.
Neither—The lock value block of the process is not written; the resource lock value block is not returned.
Write—The contents of the process's lock value block are written to the resource lock value block.

Note that when protected write or exclusive mode locks are dequeued using the Dequeue Lock Request (SYS$DEQ) system service and the address of a lock value block is specified in the **valblk** argument, the contents of that lock value block are written to the resource lock value block.

15.5 Dequeuing Locks

When a process no longer needs a lock on a resource, you can dequeue the lock by using the Dequeue Lock Request (SYS$DEQ) system service. Dequeuing locks means that the specified lock request is removed from the queue it is in. Locks are dequeued from any queue: Granted, Waiting, or Conversion (see Section 15.2.6). When the last lock on a resource is dequeued, the lock management services delete the name of the resource from its data structures.

The four arguments to the SYS$DEQ macro (**lkid**, **valblk**, **acmode**, and **flags**) are optional. The **lkid** argument allows the process to specify a particular lock to be dequeued, using the lock identification returned in the lock status block.

The **valblk** argument contains the address of a 16-byte lock value block. If the lock being dequeued is in protected write or exclusive mode, the contents of the lock value block are stored in the lock value block associated with the resource. If the lock being dequeued is in any other mode, the lock value block is not used. The lock value block can be used only if a particular lock is being dequeued.

Three flags are available:

- LCK$M_DEQUALL—The LCK$M_DEQALL flag indicates that all locks of the access mode specified with the **acmode** argument and less privileged access modes are to be dequeued. The access mode is maximized with the access mode of the caller. If the flag LCK$M_DEQALL is specified, then the **lkid** argument must be 0 (or not specified).

- LCK$M_CANCEL—When LCK$M_CANCEL is specified, SYS$DEQ attempts to cancel a lock conversion request that was queued by SYS$ENQ. This attempt can succeed only if the lock request has not yet been granted, in which case the request is in the conversion queue. The LCK$M_CANCEL flag is ignored if the LCK$M_DEQALL flag is specified. For more information about the LCK$M_CANCEL flag, see the description of the SYS$DEQ service in the *OpenVMS System Services Reference Manual*.
- LCK$M_INVVALBLK—When LCK$M_INVVALBLK is specified, $DEQ marks the lock value block, which is maintained for the resource in the lock database, as invalid. See the descriptions of SYS$DEQ and SYS$ENQ in the *OpenVMS System Services Reference Manual* for more information about the LCK$M_INVVALBLK flag.

The following is an example of dequeuing locks:

```
#include <stdio.h>
#include <descrip.h>
#include <lckdef.h>

/* Declare a lock status block */

struct lock_blk {
                unsigned short lkstat ,reserved;
                unsigned int lock_id;
}lksb;

   .
   .
   .
        void read_updates();
        unsigned int status, lkmode=LCK$K_CRMODE, lkid;
        $DESCRIPTOR(resnam,"STRUCTURE_1"); /* resource */

/* Queue a request for concurrent read mode lock */
        status = SYS$ENQW(0,          /* efn - event flag */
                        lkmode,       /* lkmode - lock mode */
                        &lksb,        /* lksb - lock status block */
                        0,            /* flags */
                        &resnam,      /* resnam - name of resource */
                        0,            /* parid - lock id of parent */
                        &read_updates,/* astadr - AST routine */
                        0, 0, 0, 0);
        if((status & 1) != 1)
                LIB$SIGNAL(status);

   .
   .
   .
        lkid = lksb.lock_id;
        status = SYS$DEQ( lkid,        /* lkid - id of lock to be dequeued */
                       0, 0, 0);
        if((status & 1) != 1)
                LIB$SIGNAL(status);

}
```

User-mode locks are automatically dequeued when the image exits.

15.6 Local Buffer Caching with the Lock Management Services

The lock management services provide methods for applications to perform **local buffer caching** (also called distributed buffer management). Local buffer caching allows a number of processes to maintain copies of data (disk blocks, for example) in buffers local to each process and to be notified when the buffers contain invalid data because of modifications by another process. In applications where modifications are infrequent, substantial I/O can be saved by maintaining local copies of buffers. Either the lock value block or blocking ASTs (or both) can be used to perform buffer caching.

15.6.1 Using the Lock Value Block

To support local buffer caching using the lock value block, each process maintaining a cache of buffers maintains a null mode lock on a resource that represents the current contents of each buffer. (For this discussion, assume that the buffers contain disk blocks.) The value block associated with each resource is used to contain a disk block "version number." The first time a lock is obtained on a particular disk block, the current version number of that disk block is returned in the lock value block of the process. If the contents of the buffer are cached, this version number is saved along with the buffer. To reuse the contents of the buffer, the null lock must be converted to protected read mode or exclusive mode, depending on whether the buffer is to be read or written. This conversion returns the latest version number of the disk block. The version number of the disk block is compared with the saved version number. If they are equal, the cached copy is valid. If they are not equal, a fresh copy of the disk block must be read from disk.

Whenever a procedure modifies a buffer, it writes the modified buffer to disk and then increments the version number before to converting the corresponding lock to null mode. In this way, the next process that attempts to use its local copy of the same buffer finds a version number mismatch and must read the latest copy from disk rather than use its cached (now invalid) buffer.

15.6.2 Using Blocking ASTs

Blocking ASTs are used to notify processes with granted locks that another process with an incompatible lock mode has been queued to access the same resource.

Blocking ASTs can be used to support local buffer caching in two ways. One technique involves deferred buffer writes; the other technique is an alternative method of local buffer caching without using value blocks.

15.6.2.1 Deferring Buffer Writes

When local buffer caching is being performed, a modified buffer must be written to disk before the exclusive mode lock can be released. If a large number of modifications are expected (particularly over a short period of time), you can reduce disk I/O by maintaining the exclusive mode lock for the entire time that the modifications are being made and by writing the buffer once. However, this prevents other processes from using the same disk block during this interval. This problem can be avoided if the process holding the exclusive mode lock has a blocking AST. The AST notifies the process if another process needs to use the same disk block. The holder of the exclusive mode lock can then write the buffer to disk and convert its lock to null mode (thereby allowing the other process to access the disk block). However, if no other process needs the same disk block, the first process can modify it many times but write it only once.

15.6.2.2 Buffer Caching

To perform local buffer caching using blocking ASTs, processes do not convert their locks to null mode from protected read or exclusive mode when finished with the buffer. Instead, they receive blocking ASTs whenever another process attempts to lock the same resource in an incompatible mode. With this technique, processes are notified that their cached buffers are invalid as soon as a writer needs the buffer, rather than the next time the process tries to use the buffer.

15.6.3 Choosing a Buffer-Caching Technique

The choice between using version numbers or blocking ASTs to perform local buffer caching depends on the characteristics of the application. An application that uses version numbers performs more lock conversions, whereas one that uses blocking ASTs delivers more ASTs. Note that these techniques are compatible; some processes can use one technique, and other processes can use the other at the same time. Generally, blocking ASTs are preferable in a low-contention environment, whereas version numbers are preferable in a high-contention environment. You can even invent combined or adaptive strategies.

In a **combined** strategy, the applications use specific techniques. If a process is expected to reuse the contents of a buffer in a short amount of time, the application uses blocking ASTs; if there is no reason to expect a quick reuse, the application uses version numbers.

In an **adaptive** strategy, an application makes evaluations based on the rate of blocking ASTs and conversions. If blocking ASTs arrive frequently, the application changes to using version numbers; if many conversions take place and the same cached copy remains valid, the application changes to using blocking ASTs.

For example, suppose one process continually displays the state of a database, while another occasionally updates it. If version numbers are used, the displaying process must always make sure that its copy of the database is valid (by performing a lock conversion); if blocking ASTs are used, the display process is informed every time the database is updated. On the other hand, if updates occur frequently, the use of version numbers is preferable to continually delivering blocking ASTs.

15.7 Example of Using Lock Management Services

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 both the status of the system service and the condition value returned in the lock status block to ensure that the request completed successfully.

```
! Define lock modes
INCLUDE '($LCKDEF)'
! Define lock status block
INTEGER*2 LOCK_STATUS,
2         NULL
INTEGER LOCK_ID
COMMON /LOCK_BLOCK/ LOCK_STATUS,
2                   NULL,
2                   LOCK_ID
              .
              .
              .
! Request a null lock
STATUS = SYS$ENQW (,
2                  %VAL(LCK$K_NLMODE),
2                  LOCK_STATUS,
2                  ,
2                  'TERMINAL',
2                  ,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. LOCK_STATUS) CALL LIB$SIGNAL (%VAL(LOCK_STATUS))

! Convert the lock to an exclusive lock
STATUS = SYS$ENQW (,
2                  %VAL(LCK$K_EXMODE),
2                  LOCK_STATUS,
2                  %VAL(LCK$M_CONVERT),
2                  'TERMINAL',
2                  ,,,,,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
IF (.NOT. LOCK_STATUS) CALL LIB$SIGNAL (%VAL(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.

Because the lock manager is effective only 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 **num** argument of LIB$SPAWN). Before exiting from the parent program, use SYS$WAITFR to ensure that the 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 service to write to the terminal.

16
Image Initialization

This chapter describes the system declaration mechanism, including LIB$INITIALIZE, which performs calls to any initialization routine declared for the image by the user. However, use of LIB$INITIALIZE is discouraged and should be used only when no other method is suitable. This chapter contains the following sections:

Section 16.1 describes the steps to perform image initialization.

Section 16.2 describes the argument list that is passed from the command interpreter, the debugger, or LIB$INITIALIZE to the main program.

Section 16.3 describes how a library or user program can declare an initialization routine.

Section 16.4 describes how the LIB$INITIALIZE dispatcher calls the initialization routine in a list.

Section 16.5 describes the options available to an initialization routine.

Section 16.6 illustrates with a code example several functions of an initialization routine on both VAX and Alpha systems.

16.1 Initializing an Image

In most cases, both user and library routines are self-initializing. This means that they can process information with no special action required by the calling program. Initialization is automatic in two situations:

- When the routine's statically allocated data storage is initialized at compile or link time.
- When a statically allocated flag is tested and set on each call so that initialization occurs only on the first call.

Any special initialization, such as a call to other routines or to system services, can be performed on the first call before the main program is initialized. For example, you can establish a new environment to alter the way errors are handled or the way messages are printed.

Such special initialization is required only rarely; however, when it is required, the caller of the routine does not need to make an explicit initialization call. The run-time library provides a system declaration mechanism that performs all such initialization calls before the main program is called. Thus, special initialization is invisible to later callers of the routine.

VAX

On VAX systems, before the main program or main routine is called, a number of system initialization routines are called as specified by a 1-, 2-, or 3-longword initialization list set up by the linker. ♦

Alpha

On Alpha systems, before the main program or main routine is called, a number of system initialization routines are called as specified by a 1-, 2-, or 3-quadword initialization list set up by the linker. ♦

VAX

On VAX systems, the initialization list consists of the following (in order):

- The addresses of the debugger (if present)
- The LIB$INITIALIZE routine (if present)
- The entry point of the main program or main routine ♦

Alpha

On Alpha systems, the initialization list consists of the following (in order):

- The procedure value addresses of the debugger (if present)
- The LIB$INITIALIZE routine (if present)
- The entry point of the main program or main routine ♦

The following initialization steps take place:

1. The image activator maps the user program into the address space of the process and sets up useful information, such as the program name. Then it starts the command language interpreter (CLI).
2. The CLI sets up an argument list and calls the next routine in the initialization list (debugger, LIB$INITIALIZE, main program, or main routine).
3. VAX On VAX systems, the debugger, if present, initializes itself and calls the next routine in the initialization list (LIB$INITIALIZE, main program, or main routine). ♦

   Alpha On Alpha systems, the CLI calls the debugger, if present, to set the initial breakpoints. Then the CLI calls the next entry in the vector. ♦
4. The LIB$INITIALIZE library routine, if present, calls each library and user initialization routine declared using the system LIB$INITIALIZE mechanism. Then it calls the main program or main routine.
5. The main program or main routine executes and, at the user's discretion, accesses its argument list to scan the command or to obtain information about the image. The main program or main routine can then call other routines.
6. Eventually, the main program or main routine terminates by executing a return instruction (RET) with R0 set to a standard completion code to indicate success or failure, where bit <0> equals 1 (success) or 0 (failure).
7. The completion code is returned to LIB$INITIALIZE (if present), the debugger (if present), and, finally, to the CLI, which issues a SYS$EXIT system service with the completion status as an argument. Any declared exit handlers are called at this point.

Note

Main programs should not call the SYS$EXIT system service directly. If they do, other programs cannot call them as routines.

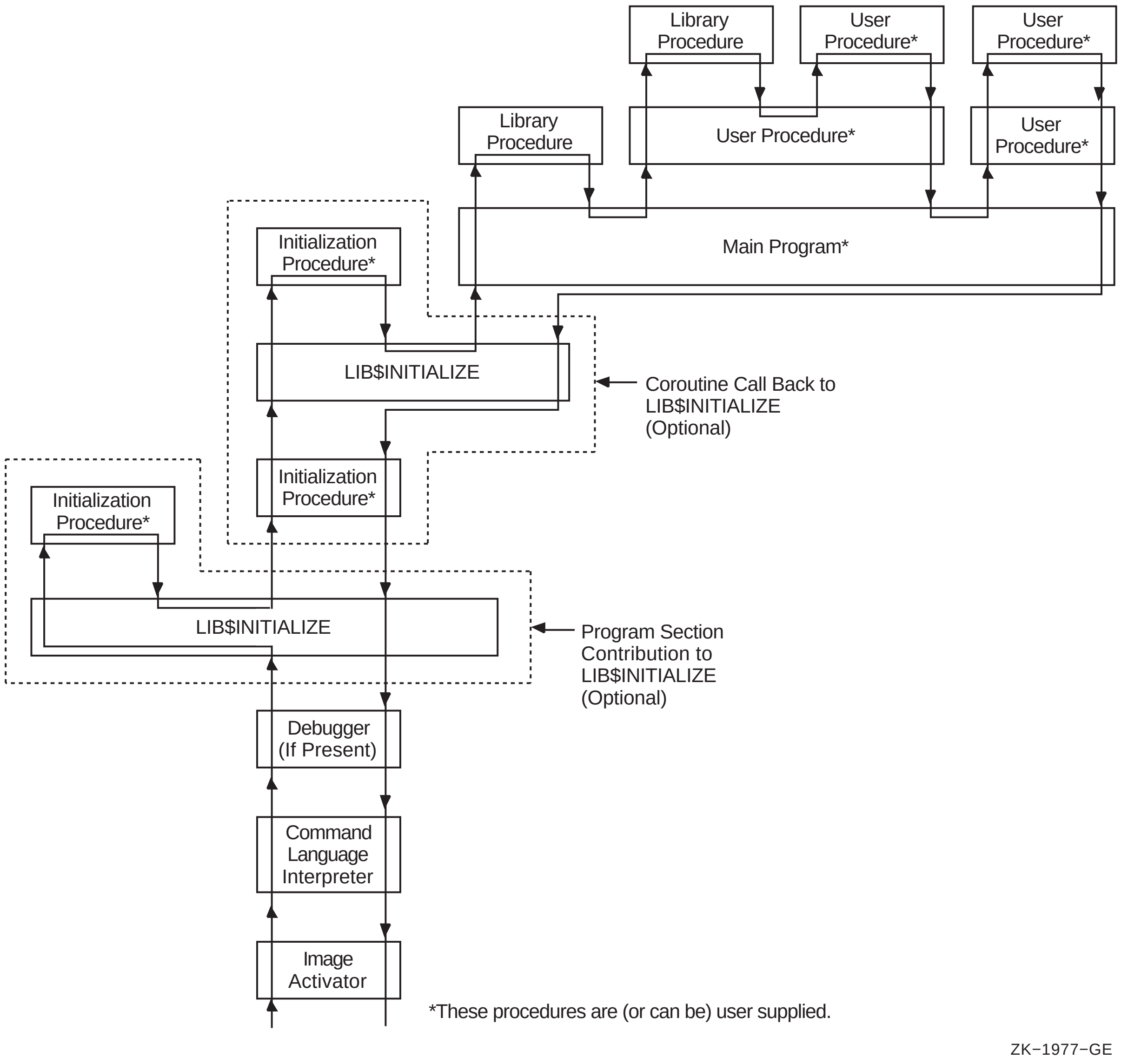
Figure 16–1 and Figure 16–2 illustrate the sequence of calls and returns in a typical image initialization. Each box is a routine activation as represented on the image stack. The top of the stack is at the top of the figure. Each upward arrow represents the result of a call instruction that creates a routine activation on the stack to which control is being transferred. Each downward arrow represents the result of a RET (return) instruction. A RET instruction removes the routine activation from the stack and causes control to be transferred downward to the next box.

A user program can alter the image initialization sequence by making a program section (PSECT) contribution to PSECT LIB$INITIALIZE and by declaring EXTERNAL LIB$INITIALIZE. This adds the optional initialization steps shown in Figure 16–1 and Figure 16–2 labeled “Program Section Contribution to LIB$INITIALIZE.” (A program section is a portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.) If the initialization routine also performs a coroutine call back to LIB$INITIALIZE, the optional steps labeled “Coroutine Call Back to LIB$INITIALIZE” in Figure 16–1 and Figure 16–2 are added to the image initialization sequence.

VAX

Figure 16–1 shows the call instruction calling the debugger, if present, and the debugger then directly calling LIB$INITIALIZE and the main program. ♦

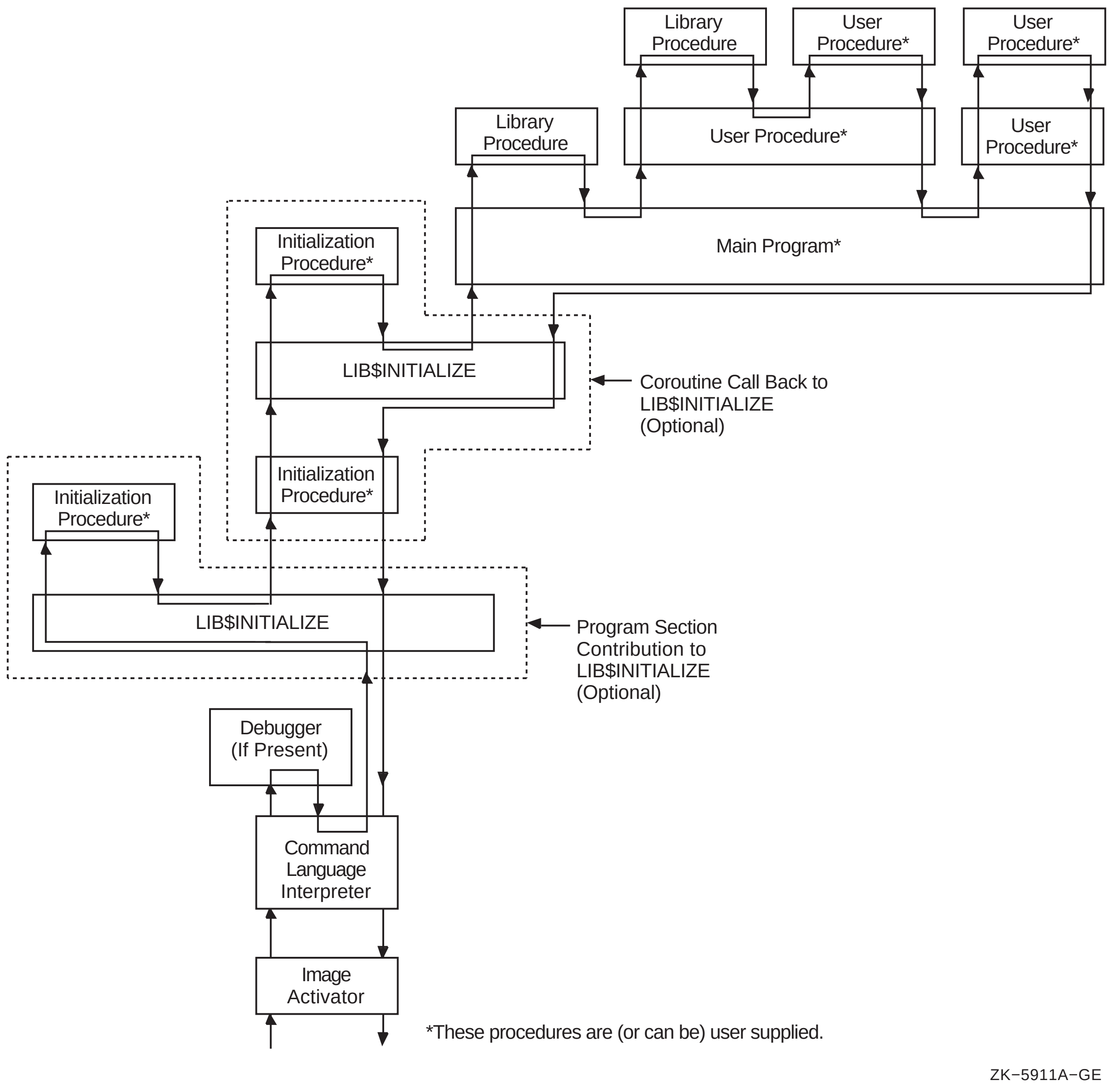
Figure 16–1 Sequence of Events During Image Initialization on VAX Systems


Library Procedure
User Procedure*
User Procedure*
Library Procedure
User Procedure*
User Procedure*
Initialization Procedure*
Main Program*
LIB$INITIALIZE
Coroutine Call Back to LIB$INITIALIZE (Optional)
Initialization Procedure*
Initialization Procedure*
LIB$INITIALIZE
Program Section Contribution to LIB$INITIALIZE (Optional)
Debugger (If Present)
Command Language Interpreter
Image Activator
*These procedures are (or can be) user supplied.
ZK−1977−GE

Alpha

Figure 16–2 shows the call instruction calling the debugger, if present, to set a breakpoint at the main program's entry point. ♦

Figure 16–2 Sequence of Events During Image Initialization on Alpha Systems


Library Procedure
User Procedure*
User Procedure*
Library Procedure
User Procedure*
User Procedure*
Main Program*
Initialization Procedure*
LIB$INITIALIZE
Coroutine Call Back to LIB$INITIALIZE (Optional)
Initialization Procedure*
Initialization Procedure*
LIB$INITIALIZE
Program Section Contribution to LIB$INITIALIZE (Optional)
Debugger (If Present)
Command Language Interpreter
Image Activator
*These procedures are (or can be) user supplied.
ZK−5911A−GE

16.2 Initializing an Argument List

The following argument list is passed from the CLI, the debugger, or LIB$INITIALIZE to the main program. This argument list is the same for each routine activation.

(start ,cli-coroutine [,image-info])

The **start** argument is the address of the entry in the initialization vector that is used to perform the call.

The **cli-routine** argument is the address of a CLI coroutine to obtain command arguments. For more information, see the *OpenVMS Utility Routines Manual*.

The **image-info** argument is useful image information, such as the program name.

The debugger or LIB$INITIALIZE, or both, can call the next routine in the initialization chain using the following coding sequence:

```
   .
   .
   .
ADDL     #4, 4(AP)      ; Step to next initialization list entry
MOVL     @4(AP), R0     ; R0 = next address to call
CALLG    (AP), (R0)     ; Call next initialization routine
   .
   .
   .
```

This coding sequence modifies the contents of an argument list entry. Thus, the sequence does not follow the OpenVMS calling standard. However, the argument list can be expanded in the future without requiring any change either to the debugger or to LIB$INITIALIZE.

16.3 Declaring Initialization Routines

Any library or user program module can declare an initialization routine. This routine is called when the image is started. The declaration is made by making a contribution to the LIB$INITIALIZE program section, which contains a list of routine entry point addresses to be called before the main program or main routine is called.

The following VAX MACRO example declares an initialization routine by placing the routine entry address INIT_PROC in the list:

```
.EXTRN LIB$INITIALIZE          ; Cause library initialization
                               ; Dispatcher to be loaded

.PSECT LIB$INITIALIZE, NOPIC, USR, CON, REL, GBL, NOSHR, NOEXE, RD, NOWRT, LONG

.LONG INIT_PROC                ; Contribute entry point address of
                               ; initialization routine.
.PSECT ...
```

The .EXTRN declaration links the initialization routine dispatcher, LIB$INITIALIZE, into your program's image. The reference contains a definition of the special global symbol LIB$INITIALIZE, which is the routine entry point address of the dispatcher. The linker stores the value of this special global symbol in the initialization list along with the starting address of the debugger and the main program. The GBL specification ensures that the PSECT LIB$INITIALIZE contribution is not affected by any clustering performed by the linker.

16.4 Dispatching to Initialization Routines

The LIB$INITIALIZE dispatcher calls each initialization routine in the list with the following argument list:

CALL init-proc (init-coroutine ,cli-coroutine [, image-info])

The **init-coroutine** argument is the address of a library coroutine to be called to effect a coroutine linkage with LIB$INITIALIZE.

The **cli-coroutine** is the address of a CLI coroutine used to obtain command arguments.

The **image-info** argument is useful image information, such as the program name.

16.5 Initialization Routine Options

An initialization routine can be used to do the following:

- Set up an exit handler by calling the Declare Exit Handler ($DCLEXH) system service, although exit handlers are generally set up by using a statically allocated first-time flag.
- Initialize statically allocated storage, although this is done preferably at image activation time using compile-time and link-time data initialization declarations or by using a first-time call flag in its statically allocated storage.
- Call the initialization dispatcher (instead of returning to it) by calling the **init-coroutine** argument. This achieves a coroutine link. Control returns to the initialization routine when the main program returns control. Then the initialization routine should also return control to pass back the completion code returned by the main program (to the debugger or CLI, or both).
- Establish a condition handler in the current frame before performing the preceding actions. This leaves the initialization routine condition handler on the image stack for the duration of the image execution. This occurs after the CLI sets up the catchall stack frame handler and after the debugger sets up its stack frame handler. Thus, the initialization routine handler can override either of these handlers, because it will receive signals before they do.

16.6 Initialization Example

The following VAX MACRO code fragment, which works on both VAX and Alpha systems, shows how an initialization routine does the following:

- Establishes a handler
- Calls the **init-coroutine** argument routine, so that the coroutine calls the initialization dispatcher
- Gains control after the main program returns
- Returns to the normal exit processing

```
      .ENTRY INIT_PROC, ^M<>     ; No registers used
      MOVAL HANDLER, (FP)        ; Establish handler
      ...                        ; Perform any other initialization

      CALLG (AP), @INIT_CO_ROUTINE(AP)
                                 ; Continue initialization which
10$:                             ; then calls main program or
                                 ; routine.
     ...                         ; Return here when main program
                                 ; returns with R0 = completion
     RET                         ; Status return to normal exit
                                 ; processing with R0 = completion
                                 ; status

     .ENTRY HANDLER, ^M<...>     ; Register mask
     ...                         ; handle condition
                                 ; could unwind to 10$
     MOVL #..., R0               ; Set completion status with a
                                 ; condition value
     RET                         ; Resignal or continue depending
                                 ; on R0 being SS$_RESIGNAL or
                                 ; SS$_CONTINUE.
```

17

Shareable Resources

This chapter describes the techniques available for sharing data and program code among programs. It contains the following sections:

Section 17.1 describes how to share code among programs.

Section 17.2 describes shareable images.

Section 17.3 defines and describes using local and global symbols to share images.

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

- DCL symbols and logical names
- Libraries
- Shareable images
- Global sections
- Common blocks installed in a shareable image
- OpenVMS Record Management Services (RMS) shared files

Symbols and logical names are also used for intraprocess and interprocess communication; therefore, they are discussed in Chapter 10.

Libraries and shareable images are used for sharing program code.

Global sections, common blocks stored in shareable images, and RMS shared files are used for sharing data. You can also use common blocks for interprocess communication. For more information, refer to Chapter 2.

17.1 Sharing Program Code

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

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

17.1.1 Object Libraries

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

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

17.1.1.1 System- and User-Defined Default Object Libraries

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

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

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

17.1.1.2 How the Linker Searches Libraries

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

17.1.1.3 Creating an Object Library

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

```
$ LIBRARY/CREATE INCOME
```

17.1.1.4 Managing an Object Library

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

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

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

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

Use the /DELETE qualifier to delete a library module and the /EXTRACT qualifier to recreate an object file. If you delete many modules, you should also compress (/COMPRESS qualifier) 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.

17.1.2 Text and Macro Libraries

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

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

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

You also use LIBRARIAN to create and manage text and macro libraries. Refer to Section 17.1.1.3 and Section 17.1.1.4 for a summary of LIBRARIAN commands.

17.2 Shareable Images

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

- Saves disk space—The executable images to which the shareable image is linked do not physically include the shareable image. Only one copy of the shareable image exists.
- Simplifies maintenance—If you use transfer vectors and the GSMATCH (on VAX systems) or symbol vectors (on Alpha systems) option, 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 the *OpenVMS Linker Utility Manual* for more information about creating shareable images and shareable image libraries.

17.3 Symbols

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

17.3.1 Defining Symbols

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

17.3.2 Local and Global Symbols

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

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

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

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

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

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

17.3.3 Resolving Global Symbols

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

1. Explicitly named modules and libraries—Generally used to resolve user-defined global symbols, such as subprogram names and condition codes. These modules and libraries are searched in the order in which they are specified.
2. System default libraries—Generally used to resolve system-defined global symbols, such as procedure names and condition codes.
3. User default libraries—Generally used to avoid explicitly naming libraries, thereby simplifying linking.

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

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

17.3.3.1 Explicitly Named Modules and Libraries

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

```
$ LINK INCOME, GETSTATS
```

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

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

17.3.3.2 System Default Libraries

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

17.3.3.3 User Default Libraries

If you write general-purpose procedures or define general-purpose symbols, you can place them in a user default library. (You can also make your development library a user default library.) In this way, you can link to the modules containing these procedures and symbols without explicitly naming the library in the DCL LINK command. To name a single-user library, equate the file name of the library to the logical name LNK$LIBRARY. For subsequent default libraries, use the logical names LNK$LIBRARY_1 through LNK$LIBRARY_999, as described in Section 17.1.1.

17.3.3.4 Making a Library Available for Systemwide Use

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

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

17.3.3.5 Macro Libraries

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

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

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

LBRDEF.MAR

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

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

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

The following LINK command uses the object file directly:

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

17.3.4 Sharing Data

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

17.3.4.1 Installed Common Blocks

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

To install a common block as a shared image:

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

   Alpha

   For Alpha only, you need to specify a Linker options file (shown here as SYS$INPUT to allow typed input) to specify the PSECT attributes of the COMMON block PSECT and include it in the global symbol table:

   ```
   $ LINK/SHAREABLE INC_COMMON ,SYS$INPUT/OPTION
   _ SYMBOL_VECTOR=(WORK_AREA=PSECT)
   _ PSECT_ATTR=WORK_AREA,SHR
   ```

   With DEC Fortran 90 on OpenVMS Alpha systems, the default PSECT attribute for a common block is NOSHR. To use a shared installed common block, you *must* specify one of the following:

   - The SHR attribute in a cDEC$ PSECT directive in the source file
   - The SHR attribute in the Linker options file for the shareable image to be installed and for each executable image that references the installed common block

   If the !DEC$ PSECT (same as cDEC$ PSECT) directive specified the SHR attribute, the LINK command is as follows:

   ```
   $ LINK/SHAREABLE INC_COMMON  ,SYS$INPUT/OPTION
   _ SYMBOL_VECTOR=(WORK_AREA=PSECT)
   ```

   For Alpha only, copy the shareable image. Once created, you should copy the shareable image into SYS$SHARE before it is installed. The file protection of the .EXE file must allow write access for the processes running programs that will access the shareable image (shown for Group access in the following COPY command):

   ```
   $ COPY/LOG DISK$:[INCOME.DEV]INC_COMMON.EXE SYS$SHARE:*.*
   _ /PROTECTION=G:RWE
   ```

   If you do not copy the installed shareable image to SYS$SHARE, before running executable images that reference the installed shareable common image, you must define a logical name that specifies the location of that image.

   When compiling the program that contains the common block declarations, consistently use the *same* /ALIGNMENT and /GRANULARITY qualifiers used to compile the common block data declaration program that has been installed as a shareable image. For more information, see Section 17.3.4.3. ♦

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, enter CREATE, followed by the complete file specification of the shareable image that contains the common block (the file type defaults to .EXE) and the qualifiers /WRITEABLE and /SHARED. The Install utility installs your shareable image and reissues the INSTALL prompt. Enter EXIT to exit. Remember to remove CMKRNL privilege. (For complete documentation of the Install utility, see the *OpenVMS System Management Utilities Reference Manual*.)

The following example shows how to install a shareable image:

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

Note

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

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

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

   Alpha For Alpha only, when compiling the program that contains the common block declarations, consistently use the *same* /ALIGNMENT and /GRANULARITY qualifiers used to compile the common block data declaration program that has been installed as a shareable image. For more information, see Section 17.3.4.3. ♦
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
   ```

   Alpha For Alpha only, linking is as follows:

   ```
   INC_COMMON/SHAREABLE
   PSECT_ATTR=WORK_AREA, SHR
   ```

   If a !DEC$ PSECT (cDEC$ PSECT) directive specified the SHR PSECT attribute, the linker options file INCOME.OPT would contain the following line:

   ```
   INC_COMMON/SHAREABLE
   ```

   The source line containing the !DEC$ PSECT directive would be as follows:

   ```
   !DEC$ PSECT /INC_COMMON/ SHR
   ```
   ♦
4. Execute the program.

Alpha

For Alpha only, if the installed image is not located in SYS$SHARE, you must define a logical name that specifies the location of that image. The logical name (in this example INC_COMMON) is the name of the installed base. ♦

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 that access shared data use common event flag clusters to synchronize read and write access to the data. Refer to Chapter 15 for more information about using event flags for program synchronization.

17.3.4.2 Using Global Sections

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

To create a global section, invoke SYS$CRMPSC and add the following:

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

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

The format for SYS$MGBLSC is as follows:

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

Refer to the *OpenVMS System Services Reference Manual* for complete information about this system service.

In Example 17–1, one image, DEVICE.FOR, passes device names to another image, GETDEVINF.FOR. GETDEVINF.FOR returns the process name and the terminal associated with the process that allocated each device. The two processes use the global section GLOBAL_SEC to communicate. GLOBAL_SEC is mapped to the common block named DATA, which is page aligned by the options file DATA.OPT. Event flags are used to synchronize the exchange of information. UFO_CREATE.FOR, DATA.OPT, and DEVICE.FOR are included here for easy reference. Refer to Section 8.4 for additional information about global sections.

Example 17–1 Interprocess Communication Using Global Sections

```
!UFO_CREATE.FOR
   .
   .
   .
INTEGER FUNCTION UFO_CREATE (FAB,
2                            RAB,
2                            LUN)
```

(continued on next page)

Example 17–1 (Cont.) Interprocess Communication Using Global Sections

```
! 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/
```

(continued on next page)

Example 17–1 (Cont.) Interprocess Communication Using Global Sections

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

(continued on next page)

Example 17–1 (Cont.) Interprocess Communication Using Global Sections

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 the 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.

17.3.4.3 Synchronizing Access to Global Sections

Alpha

If more than one process or thread will write to a shared global section containing COMMON block data, the user program may need to synchronize access to COMMON block variables.

Compile all programs referencing the shared common area with the same value for the /ALIGNMENT and /GRANULARITY qualifiers, as shown in the following:

```
$ F90 /ALIGN=COMMONS=NATURAL /GRANULARITY=LONGWORD INC_COMMON
```

Using /GRANULARITY=LONGWORD for 4-byte variables or /GRANULARITY=QUADWORD for 8-byte variables ensures that adjacent data is not accidentally effected. To ensure access to 1-byte variables, specify /GRANULARITY=BYTE. Because accessing data items less than four bytes slows run-time performance, you might want to considering synchronizing read and write access to the data on the same node. ♦

One way for programs accessing shared data is to use common event flag clusters to synchronize read and write access to the data on the same node. 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 must examine the common event flag cluster to ensure that accessing the data does not conflict with an operation already in progress.

Other ways of synchronizing access on a single node include using the following OpenVMS system services:

- The lock manager system services (SYS$ENQ and SYS$DEQ)
- The hibernate and wake system services (SYS$HIBER and SYS$WAKE)

You could also use Assembler code for synchronization.

17.3.4.4 RMS Shared Files

RMS allows concurrent access to a file. Shared files can be one of the following formats:

- Indexed files
- Relative files
- Sequential files with 512-byte fixed-length records

To coordinate access to a file, RMS uses the lock manager. You can override the RMS lock manager by controlling access yourself. Refer to Chapter 15 for more information about synchronizing access to resources.

18 Creating User-Written System Services

This chapter describes how to create user-written system services. It contains the following sections:

Section 18.1 describes privileged routines and privileged shareable images.

Section 18.2 describes how to write a privileged routine.

Section 18.3 describes how to create a privileged shareable image on VAX systems.

Section 18.4 describes how to create a privileged shareable image on Alpha systems.

18.1 Overview

Your application may contain certain routines that perform privileged functions, called **user-written system services**. To create these routines, put them in a privileged shareable image. User-mode routines in other modules can call the routines in the privileged shareable image to perform functions in a more privileged mode.

You create a privileged shareable image as you would any other shareable image, using the /SHAREABLE qualifier with the linker. (For more information about how to create a shareable image, see the *OpenVMS Linker Utility Manual*.) However, because a call to a routine in a more privileged mode must be vectored through the system service dispatch routine, you must perform some additional steps. The following steps outline the basic procedure. Section 18.3 provides more detail about requirements specific to VAX systems. Section 18.4 describes the necessary steps for Alpha systems.

1. Create the source file. The source file for a privileged shareable image contains the routines that perform privileged functions. In addition, because user-written system services are called using the system service dispatcher, you must include a privileged library vector (PLV) in your shareable image. A PLV is an operating-system-defined data structure that communicates the location of the privileged routines to the operating system.

   VAX

   On VAX systems, the PLV contains the addresses of dispatch routines for each access mode used in the image. You must write these dispatch routines and include them in your shareable image. Section 18.3.1 provides more information. ♦

Alpha

On Alpha systems, you list the names of the privileged routines in the PLV, sorted by access mode. You do not need to create dispatch routines; the image activator creates them for you automatically. ♦

Section 18.2 provides guidelines for creating privileged routines.

2. Compile or assemble the source file.
3. Create the shareable image. You create a privileged shareable image as you would any other shareable image: by specifying the /SHAREABLE qualifier to the LINK command. Note, however, that creating privileged shareable images has some additional requirements. The following list summarizes these requirements. See the *OpenVMS Linker Utility Manual* for additional information about linker qualifiers and options.
   - Declare the privileged routine entry points as universal symbols. Privileged shareable images use the same mechanisms to declare universal symbols as other shareable images: transfer vectors on VAX and symbol vectors on Alpha systems. However, because calls to user-written system services must be vectored through the system service dispatcher, you must use extensions to these mechanisms for privileged shareable images. Section 18.3.3 describes how to declare a universal symbol in a VAX privileged shareable image. Section 18.4.2 describes how to declare a universal symbol in an Alpha system privileged shareable image.
   - Prevent the linker from processing the system default shareable image library, SYS$LIBRARY:IMAGELIB.OLB, by specifying the /NOSYSSHR linker qualifier. Otherwise, the linker processes this library by default.
   - Protect the shareable image from user-mode access by specifying the /PROTECT linker qualifier. If you want to protect only certain portions of the shareable image, instead of the entire image, use the PROTECT= linker option.
   - Set the VEC attribute of the program section containing the PLV by using the PSECT_ATTR= linker option. Modules written in MACRO can specify this attribute in the .PSECT directive. The PLV must appear in a program section with the VEC attribute set.
   - Set the shareable image identification numbers using the GSMATCH= option.

   If your privileged application requires that you link against the system executive, see the *OpenVMS Linker Utility Manual* for more information.
4. Install the privileged shareable image as a protected permanent global section. Privileged shareable images must be installed to be available to nonprivileged programs. The following procedure is recommended:
   a. Move the privileged shareable image to a protected directory, such as SYS$SHARE.
   b. Invoke the Install utility, specifying the /PROTECT, /OPEN, and /SHARED qualifiers. You can also specify the /HEADER_RESIDENT qualifier. The following entry could be used to install a user-written system service whose image name is MY_PRIV_SHARE:

```
$ INSTALL
INSTALL> ADD SYS$SHARE:MY_PRIV_SHARE/PROTECT/OPEN/SHARED/HEADER_RES
```

To use a privileged shareable image, you include it in a link operation as you would any other shareable image: specifying the shareable image in a linker options file with the /SHAREABLE qualifier appended to the file specification to identify it as a shareable image.

18.2 Writing a Privileged Routine (User-Written System Service)

On both VAX systems and Alpha systems, the routines that implement user-written system services must enable any privileges they need that the nonprivileged user of the user-written system service lacks. The user-written system service must also disable any such privileges before the nonprivileged user receives control again. To enable or disable a set of privileges, use the Set Privileges ($SETPRV) system service. The following example shows the operator (OPER) and physical I/O (PHY_IO) privileges being enabled. (Any code executing in executive or kernel mode is granted an implicit SETPRV privilege so it can enable any privileges it needs.)

```
PRVMSK:  .LONG   <1@PRV$V_OPER>!<1@PRV$V_PHY_IO> ;OPER and PHY_IO
         .LONG   0     ;quadword mask required.  No bits set in
                       ;high-order longword for these privileges.
           .
           .
           .
        $SETPRV_S  ENBFLG=#1,-       ;1=enable, 0=disable
                   PRVADR=PRVMSK     ;Identifies the privileges
```

When you design your system service, you must carefully define the boundaries between the protected subsystem and the user who calls the service. A protected image has privileges to perform tasks on its own behalf. When your image performs tasks on behalf of the user, you must ensure that your image performs only those tasks the user could not have done on his or her own. Always keep the following coding principles in mind:

- Keep privileges off, and turn them on only when necessary.
- Make sure privileges are off on all exit paths. When you perform a task for the user, operate in user mode whenever possible and operate at all times with the user's privileges, identity, and so on. Make sure that operating in an inner mode does not give you any special privileges with respect to the operation being performed. Resume a privileged state only when you are about to resume operation on your own behalf.
- If user input can affect an operation executed with privilege, you have to carefully validate the input. Never pass user parameters directly to an operation executed in an inner mode or with privilege. When designing your program, keep in mind that the inner modes implicitly provide a user with the system privileges SETPRV, CMKRNL, SYSNAM, and SYSLCK. (See the *OpenVMS Guide to System Security* for descriptions.)
- As a protected image, your program does not have the entire operating system programming environment at its disposal. Unless a module has the prefix SYS$ or EXE$, you must avoid calling it from an inner mode. In particular, do not call LIB$GET_VM or LIB$RET_VM from an inner mode. You can call OpenVMS RMS routines from executive mode but not from kernel mode.

VAX

On VAX systems, Version 5.4 or later of the operating system, any OpenVMS RMS files that were opened with privilege from an inner mode can be left open during user execution; however, this is not acceptable on earlier versions of the operating system. ♦

- Never make subroutine calls to other shareable images from kernel or executive mode.
- When a protected subsystem opens a file on its own behalf, it should specify executive-mode logical names only by naming executive mode explicitly in the FAB$V_LNM_MODE subfield of the file access block (FAB). This prevents a user's logical name from redirecting a file specification.

VAX

On VAX systems, refer to SYS$EXAMPLES:USSDISP.MAR and USSTEST.MAR for listings of modules in a user-written system service and of a module that calls the user-written system service. ♦

Alpha

On Alpha systems, for C examples refer to SYS$EXAMPLES:UWSS.C and SYS$EXAMPLES:UWSS_TEST.C. ♦

18.3 Creating a Privileged Shareable Image (VAX Only)

VAX

On VAX systems, you must create dispatch routines that transfer control to the privileged routines in your shareable image. You then put the addresses of these dispatch routines in a privileged library vector (PLV). Section 18.3.1 describes how to create a dispatch routine. Section 18.3.2 describes how to create a PLV.

18.3.1 Creating User-Written Dispatch Routines on VAX Systems

On VAX systems, you must create kernel-mode and executive-mode dispatching routines that transfer control to the routine entry points. You must supply one dispatch routine for all your kernel mode routines and a separate routine for all the executive mode routines. The dispatcher is usually written using the CASE construct, with each routine identified by a code number. Make sure that the identification code you use in the dispatch routine and the code specified in the transfer vector identify the same routine.

The image activator, when it activates a privileged shareable image, obtains the addresses of the dispatch routines from the PLV and stores these addresses at a location known to the system service dispatcher. When a call to a privileged routine is initiated by a CHME or CHMK instruction, the system service dispatcher attempts to match the code number with a system service code. If there is no match, it transfers control to the location where the image activator has stored the address of your dispatch routines.

A dispatch routine must validate the CHMK or CHME operand identification code number, handling any invalid operands. In addition, the dispatching routine must transfer control to the appropriate routine for each identification code if the user-written system service contains functionally separate coding segments. The CASE instruction in VAX MACRO or a computed GOTO-type statement in a high-level language provides a convenient mechanism for determining where to transfer control.

Note

Users of your privileged shareable image must specify the same code number to identify a privileged routine as you used to identify it in the dispatch routine. Users specify the code number in their CHMK or CHME instruction. See Section 18.3.3 for information about transfer vectors.

In your source file, a dispatch routine must precede the routines that implement the user-written system service.

Example 18–1 illustrates a sample dispatching routine, taken from the sample privileged shareable image in SYS$EXAMPLES named USSDISP.MAR.

Example 18–1 Sample Dispatching Routine

```
KERNEL_DISPATCH::                       ; Entry to dispatcher
        MOVAB   W^-KCODE_BASE(R0),R1    ; Normalize dispatch code value
        BLSS    KNOTME                  ; Branch if code value too low
        CMPW    R1,#KERNEL_COUNTER      ; Check high limit
        BGEQU   KNOTME                  ; Branch if out of range
;
; The dispatch code has now been verified as being handled by this dispatcher,
; now the argument list will be probed and the required number of arguments
; verified.
;
        MOVZBL  W^KERNEL_NARG[R1],R1    ; Get required argument count
        MOVAL   @#4[R1],R1              ; Compute byte count including argcount
        IFNORD  R1,(AP),KACCVIO         ; Branch if arglist not readable
        CMPB    (AP),W^<KERNEL_NARG-KCODE_BASE>[R0] ; Check for required number
        BLSSU   KINSFARG                ;  of arguments
        MOVL    FP,SP                   ; Reset stack for service routine
        CASEW   R0,-                    ; Case on change mode
        .
        .
        .
```

18.3.2 Creating a PLV on VAX Systems

On VAX systems, a call to a privileged routine goes to the transfer vector which executes a change mode instruction (CHM*x*) specifying the identification code of the privileged routine as the operand to the instruction. The operating system routes the change mode instruction to the system service dispatch routine, which attempts to locate the system service with the code specified. Because the code is a negative number, the system service dispatcher drops through its list of known services and transfers control to a user-written dispatch routine, if any have been specified.

The image activator has already placed at this location the address of whatever user-written dispatch routines it found in the privileged shareable image's PLV when it activated the PLV. The dispatch routine transfers control to the routine in the shareable image identified by the code. (You must ensure that the code used in the transfer vector and the code specified in the dispatch routine both identify the same routine.) Figure 18–1 illustrates this flow of control.

Figure 18–1 Flow of Control Accessing a Privileged Routine on VAX Systems

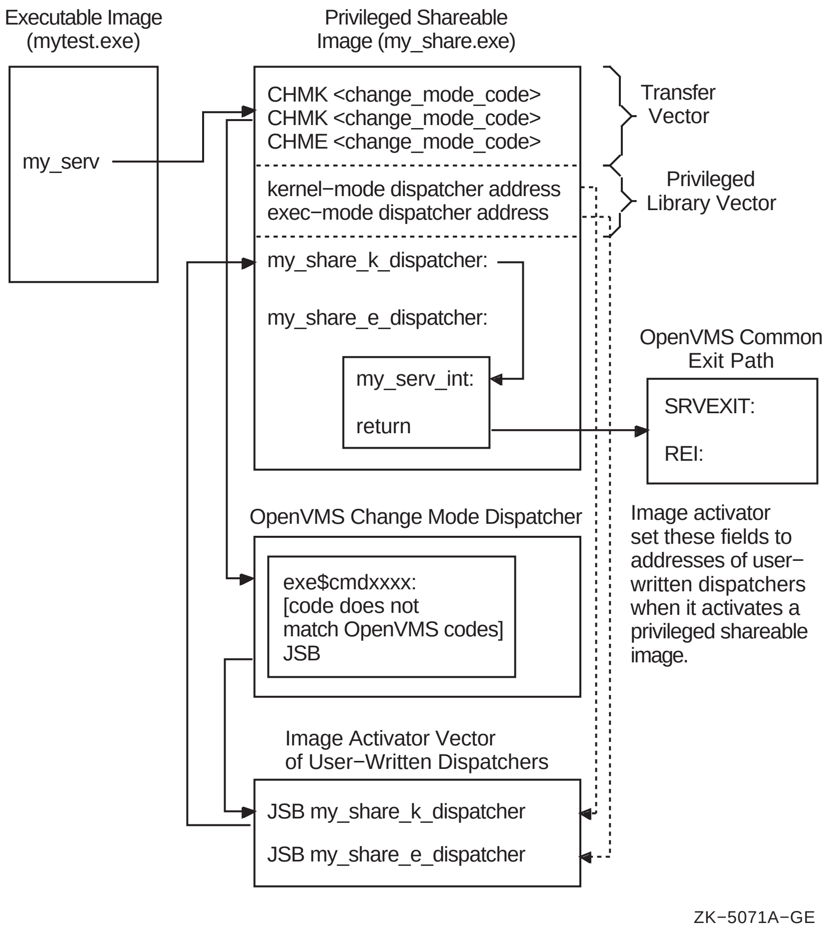

Executable Image (mytest.exe)
my_serv
Privileged Shareable Image (my_share.exe)
CHMK <change_mode_code>
CHMK <change_mode_code>
CHME <change_mode_code>
kernel−mode dispatcher address
exec−mode dispatcher address
my_share_k_dispatcher:
my_share_e_dispatcher:
my_serv_int:
return
Transfer Vector
Privileged Library Vector
OpenVMS Common Exit Path
SRVEXIT:
REI:
OpenVMS Change Mode Dispatcher
exe$cmdxxxx:
[code does not match OpenVMS codes]
JSB
Image activator set these fields to addresses of user−written dispatchers when it activates a privileged shareable image.
Image Activator Vector of User−Written Dispatchers
JSB my_share_k_dispatcher
JSB my_share_e_dispatcher
ZK−5071A−GE

Figure 18–2 shows the components of the PLV in VAX shareable images.

Figure 18–2 Components of the Privileged Library Vector on VAX Systems

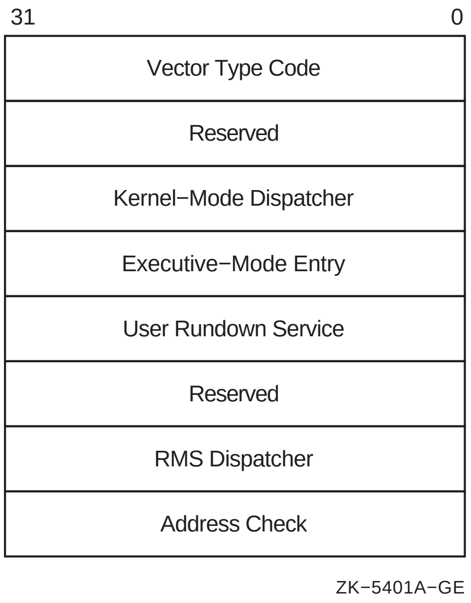


Table 18–1 describes each field in the PLV on a VAX processor, including the symbolic names the operating system defines to access each field. These names are defined by the $PLVDEF macro in SYS$LIBRARY:STARLET.MLB.

Table 18–1 Components of the VAX Privileged Library Vector

Component	Symbol	Description
Vector type code	PLV$L_TYPE	Identifies the type of vector. For PLVs, you must specify the symbolic constant defined by the operating system, PLV$C_TYP_CMOD, which identifies a privileged library vector.
Kernel-mode dispatcher	PLV$L_KERNEL	Contains the address of the user-supplied kernel-mode dispatching routine if your privileged library contains routines that run in kernel mode. The address is expressed as an offset relative to the start of the data structure (self-relative pointer). A value of 0 indicates that a kernel-mode dispatcher does not exist.
Executive-mode dispatcher	PLV$L_EXEC	Contains the address of the user-supplied executive-mode dispatching routine if your privileged library contains routines that run in executive mode. The address is expressed as an offset relative to the start of the data structure (self-relative pointer). A value of 0 indicates that a kernel-mode dispatcher does not exist.

(continued on next page)

Table 18–1 (Cont.) Components of the VAX Privileged Library Vector

Component	Symbol	Description
User-supplied rundown routine	PLV$L_USRUNDWN	Contains the address of a user-supplied rundown routine that performs image-specific cleanup and resource deallocation if your privileged library contains such a routine. When the image linked against the user-written system service is run down by the system, this run-time routine is invoked. Unlike exit handlers, the routine is always called when a process or image exits. (The image rundown code calls this routine with a JSB instruction; it returns with an RSB instruction called in kernel mode at IPL 0.)
RMS dispatcher	PLV$L_RMS	Contains the address of a user-supplied dispatcher for OpenVMS RMS services. A value of 0 indicates that a user-supplied OpenVMS RMS dispatcher does not exist. Only one user-written system service should specify the OpenVMS RMS vector, because only the last value is used. This field is intended for use only by Digital.
Address check	PLV$L_CHECK	Contains a value to verify that a user-written system service that is not position independent is located at the proper virtual address. If the image is position independent, this field should contain a zero. If the image is not position independent, this field should contain its own address.

Example 18–2 illustrates how the sample privileged shareable image in SYS$EXAMPLES assigns values to the PLV.

Example 18–2 Assigning Values to a PLV on a VAX System

```
        .PAGE
        $PLVDEF                         ; Define PLV fields
        .SBTTL  Change Mode Dispatcher Vector Block
1       .PSECT  USER_SERVICES,PAGE,VEC,PIC,NOWRT,EXE

2     .LONG   PLV$C_TYP_CMOD          ; Set type of vector to change mode
        .LONG   0                       ; Reserved
        .LONG   KERNEL_DISPATCH-.       ; Offset to kernel mode dispatcher
        .LONG   EXEC_DISPATCH-.         ; Offset to executive mode dispatcher
        .LONG   USER_RUNDOWN-.          ; Offset to user rundown service
        .LONG   0                       ; Reserved.
        .LONG   0                       ; No RMS dispatcher
        .LONG   0                       ; Address check - PIC image
```

1 The sample program sets the VEC attribute of the program section containing the PLV.

2 Values are assigned to each field of the PLV.

18.3.3 Declaring Privileged Routines as Universal Symbols Using Transfer Vectors on VAX Systems

On VAX systems, you use the transfer vector mechanism to declare universal symbols (described in the *OpenVMS Linker Utility Manual*). However, for privileged shareable images, the transfer vector must also contain a CHM*x* instruction because the target routine operates in a more privileged mode. You identify the privileged routine by its identification code, supplied as the only operand to the CHMx instruction. Note that the code number used must match the code used to identify the routine in the dispatch routine. The following example illustrates a typical transfer vector for a privileged routine:

```
.TRANSFER  my_serv
.MASK      my_serv
CHMK  <code_number>
RET
```

Because the OpenVMS system services codes are all positive numbers and because the call to a privileged routine is initially handled by the system service dispatcher, you should assign negative code numbers to identify your privileged routines so they do not conflict with system services identification codes. ♦

18.4 Creating a User-Written System Service (Alpha Only)

Alpha

On Alpha systems, in addition to the routines that perform privileged functions, you must also include a PLV in your source file. However, on Alpha systems, you list the privileged routines by name in the PLV. You do not need to create a dispatch routine that transfers control to the routine; the routine is identified by a special code.

18.4.1 Creating a PLV on Alpha Systems

On Alpha systems, the PLV contains a list of the actual addresses of the privileged routines. The image activator creates the dispatch routines. Figure 18–3 illustrates the linkage for a privileged routine on Alpha systems.

Figure 18–3 Linkage for a Privileged Routine after Image Activation

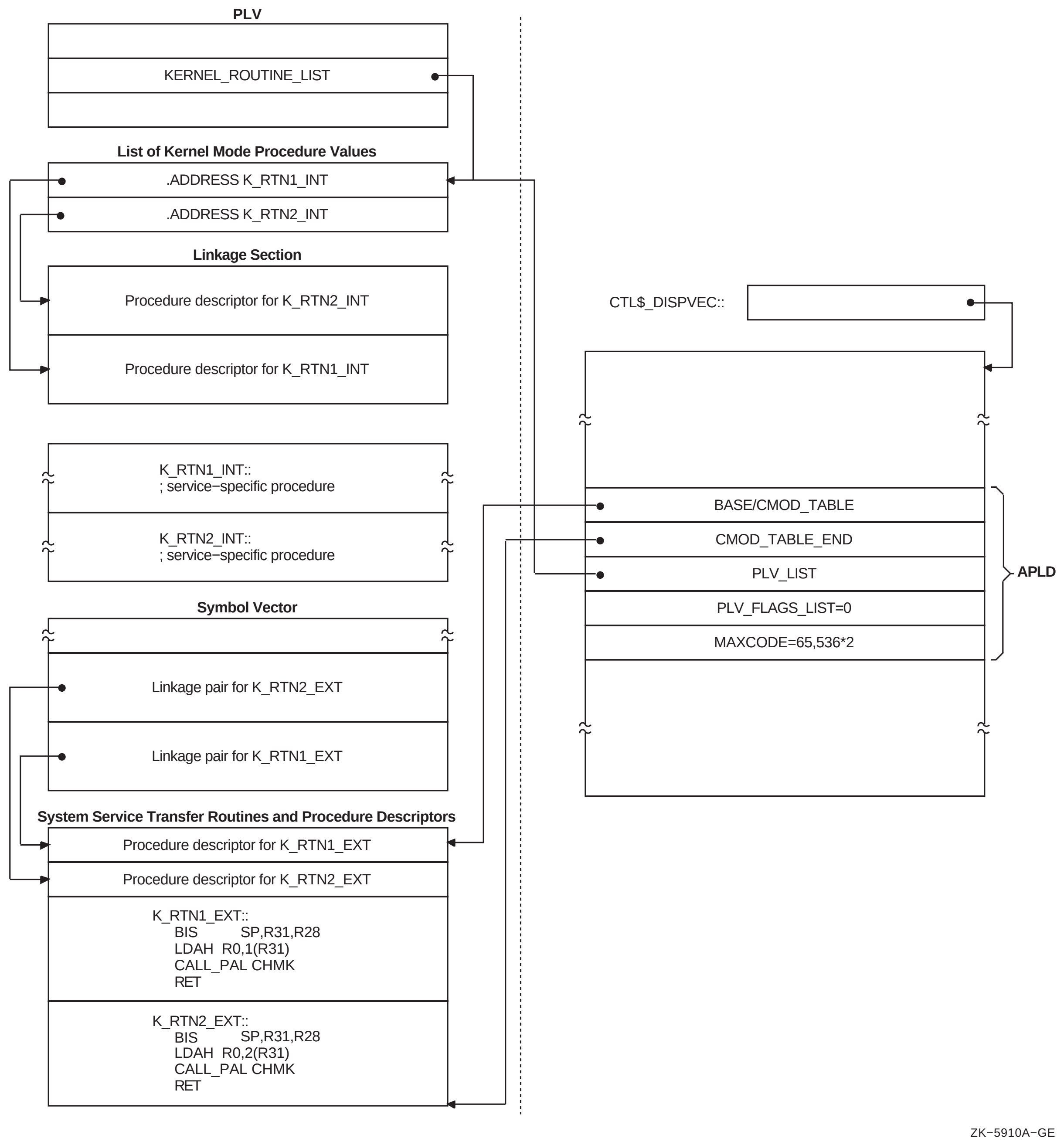

PLV
KERNEL_ROUTINE_LIST
List of Kernel Mode Procedure Values
.ADDRESS K_RTN1_INT
.ADDRESS K_RTN2_INT
Linkage Section
Procedure descriptor for K_RTN2_INT
Procedure descriptor for K_RTN1_INT
K_RTN1_INT::
; service−specific procedure
K_RTN2_INT::
; service−specific procedure
Symbol Vector
Linkage pair for K_RTN2_EXT
Linkage pair for K_RTN1_EXT
System Service Transfer Routines and Procedure Descriptors
Procedure descriptor for K_RTN1_EXT
Procedure descriptor for K_RTN2_EXT
K_RTN1_EXT::
BIS SP,R31,R28
LDAH R0,1(R31)
CALL_PAL CHMK
RET
K_RTN2_EXT::
BIS SP,R31,R28
LDAH R0,2(R31)
CALL_PAL CHMK
RET
CTL$_DISPVEC::
BASE/CMOD_TABLE
CMOD_TABLE_END
PLV_LIST
PLV_FLAGS_LIST=0
MAXCODE=65,536*2
APLD
ZK−5910A−GE

Table 18–2 describes the components of the privileged library vector on Alpha systems.

Table 18–2 Components of the Alpha Privileged Library Vector

Component	Symbol	Description
Vector type code	PLV$L_TYPE	Identifies the type of vector. You must specify the symbolic constant, PLV$C_TYP_CMOD, to identify a privileged library vector.
System version number	PLV$L_VERSION	System version number (unused).
Kernel-mode routine count	PLV$L_KERNEL_ROUTINE_COUNT	Specifies the number of user-supplied kernel-mode routines listed in the kernel-mode routine list. The address of this list is specified in PLV$PS_KERNEL_ROUTINE_LIST.
Executive-mode routine count	PLV$L_EXEC_ROUTINE_COUNT	Specifies the number of user-supplied executive-mode routines listed in the executive-mode routine list. The address of this list is specified in PLV$PS_EXEC_ROUTINE_LIST.
Kernel-mode routine list	PLV$PS_KERNEL_ROUTINE_LIST	Specifies the address of a list of user-supplied kernel-mode routines.
Executive-mode routine list	PLV$PS_EXEC_ROUTINE_LIST	Specifies the address of a list of user-supplied executive-mode routines.
User-supplied rundown routine	PLV$PS_KERNEL_RUNDOWN_HANDLER	May contain the address of a user-supplied rundown routine that performs image-specific cleanup and resource deallocation. When the image linked against the user-written system service is run down by the system, this run-time routine is invoked. Unlike exit handlers, the routine is always called when a process or image exits. (Image rundown code calls this routine with a JSB instruction; it returns with an RSB instruction called in kernel mode at IPL 0.)
Thread-safe system service	PLV$M_THREAD_SAFE	Flags the system service dispatcher that the service requires no explicit synchronization. It is assumed by the dispatcher that the service provides its own internal data synchronization and that multiple kernel threads can safely execute the service in parallel.
RMS dispatcher	PLV$PS_RMS_DISPATCHER	Address of an alternative RMS dispatching routine.
Kernel Routine Flags Vector	PLV$PS_KERNEL_ROUTINE_FLAGS	Contains either the address of an array of quadwords that contain the defined flags associated with each kernel system service, or a zero. If a flag is set, the kernel mode service may return the status SS$_WAIT_CALLERS_MODE.
Executive Routine Flags Vector	PLV$PS_EXEC_ROUTINE_FLAGS	Contains a zero value, because there are no defined flags for executive mode.

Example 18–3 illustrates how to create a PLV on Alpha systems.

Example 18–3 Creating a PLV on Alpha Systems

```
! What follows is the definition of the PLV. The PLV lives
! in its own PSECT, which must have the VEC attribute. The
! VEC attribute is forced in the linker. The PLV looks like
! this:
!
!   +--------------------------------------+
!   |          Vector Type Code            | PLV$L_TYPE
!   |          (PLV$C_TYP_CMOD)            |
!   +--------------------------------------+
!   |        System Version Number         | PLV$L_VERSION
!   |              (unused)                |
!   +--------------------------------------+
!   |      Count of Kernel Mode Services   | PLV$L_KERNEL_ROUTINE_COUNT
!   |                                      |
!   +--------------------------------------+
!   |      Count of Exec Mode Services     | PLV$L_EXEC_ROUTINE_COUNT
!   |                                      |
!   +--------------------------------------+
!   |  Address of a List of Entry Points   | PLV$PS_KERNEL_ROUTINE_LIST
!   |       for Kernel Mode Services       |
!   +--------------------------------------+
!   |  Address of a List of Entry Points   | PLV$PS_EXEC_ROUTINE_LIST
!   |        for Exec Mode Services        |
!   +--------------------------------------+
!   |         Address of Kernel Mode       | PLV$PS_KERNEL_RUNDOWN_HANDLER
!   |            Rundown Routine           |
!   +--------------------------------------+
!   |                                      | PLV$M_THREAD_SAFE
!   |                                      |
!   +--------------------------------------+
!   |       Address of Alternative RMS     | PLV$PS_RMS_DISPATCHER
!   |          Dispatching Routine         |
!   +--------------------------------------+
!   |       Kernel Routine Flags Vector    | PLV$PS_KERNEL_ROUTINE_FLAGS
!   |                                      |
!   +--------------------------------------+
!   |       Exec Routine Flags Vector      | PLV$PS_EXEC_ROUTINE_FLAGS
!   |                                      |
!   +--------------------------------------+
!
PSECT OWN = USER_SERVICES (NOWRITE, NOEXECUTE);

OWN PLV_STRUCT : $BBLOCK[PLV$C_LENGTH] INITIAL (LONG (PLV$C_TYP_CMOD,! Type
                                                    ! of vector
0,                                                  ! System version number
(KERNEL_TABLE_END - KERNEL_TABLE_START) / %UPVAL,   ! Number of kernel mode
                                                    ! services
(EXEC_TABLE_END - EXEC_TABLE_START) / %UPVAL,       ! Number of exec mode
                                                    ! services
KERNEL_TABLE_START,   ! Address of list of kernel mode service routine
EXEC_TABLE_START,     ! Address of list of exec mode service routine
RUNDOWN_HANDLER,      ! Address of list of kernel mode rundown routine
0,                    ! Reserved longword
0,                    ! Address of alternate RMS dispatcher
0,                    ! reserved
0));                  ! reserved

PSECT OWN = $OWN$;
```

18.4.2 Declaring Privileged Routines as Universal Symbols Using Symbol Vectors on Alpha Systems

On Alpha systems, you declare a user-written system service to be a universal symbol by using the symbol vector mechanism. (See the *OpenVMS Linker Utility Manual* for more information about creating symbol vectors.) However, because user-written system services must be accessed by using the privileged library vector (PLV), you must specify an alias for the user-written system service. Use the following syntax for the SYMBOL_VECTOR= option to specify an alias that can be universal:

SYMBOL_VECTOR = ([universal_alias_name/]internal_name = {PROCEDURE | | DATA})

In a privileged shareable image, calls from within the image that use the alias name result in a fixup and subsequent vectoring through the PLV, which results in a mode change. Calls from within the shareable image that use the internal name are made in the caller's mode. (Calls from external images always result in a fixup.)

The linker command procedures and options file shown in Example 18–4 illustrate how to declare universal symbols in an Alpha system privileged shareable image.

Example 18–4 Declaring Universal Symbols for Privileged Shareable Image on an Alpha System

```
$ !
$ ! Link the protected shareable image containing
$ ! the user-written system services
$ !
$ LINK  /SHARE=UWSS -
        /PROTECT -
        /MAP=UWSS -
        /SYSEXE -
        /FULL/CROSS/NOTRACE -
        UWSS, -
        SYS$INPUT:/OPTIONS

!
! Set the GSMATCH options
!
GSMATCH=LEQUAL,1,1

!
! Define transfer vectors for protected shareable image
!
SYMBOL_VECTOR = ( -
                FIRST_SERVICE   = PROCEDURE, -
                SECOND_SERVICE  = PROCEDURE, -
                THIRD_SERVICE   = PROCEDURE, -
                FOURTH_SERVICE  = PROCEDURE  -
                )

!
! Need to add the VEC attribute to the PLV psect
!
PSECT=USER_SERVICES,VEC ♦
```

19 Memory Management Services and Routines (VAX Only)

This chapter describes the use of system services and run-time routines that VAX systems use to manage memory. It contains the following sections:

Section 19.1 describes the page size on VAX systems.

Section 19.2 describes the layout of virtual address space.

Section 19.3 describes extended addressing enhancements on selected VAX systems.

Section 19.4 describes the three levels of memory allocation routines.

Section 19.5 discusses how to use system services to add virtual address space, adjust working sets, control process swapping, and create and manage sections.

19.1 Virtual Page Size

To facilitate memory protection and mapping, the virtual addresss space on VAX systems is subdivided into segments of 512-byte sizes called **pages**. (On Alpha systems, memory page sizes are much larger and vary from system to system. See Chapter 20 for information about Alpha page sizes.) Versions of system services and run-time library routines that accept page-count values as arguments interpret these arguments in 512-byte quantities. Services and routines automatically round the specified addresses to page boundaries.

19.2 Virtual Address Space

The initial size of a process's virtual address space depends on the size of the image being executed. The virtual address space of an executing program consists of the following three regions:

- Process program region (P0)

  The process program region is also referred to as P0 space. P0 space contains the instructions and data for the current image.

  Your program can dynamically allocate storage in the process program region by calling run-time library (RTL) dynamic memory allocation routines or system services.

- Process control region (P1)

  The process control region is also referred to as P1 space. P1 space contains system control information and the user-mode process stack. The user mode stack expands as necessary toward the lower-addressed end of P1 space.

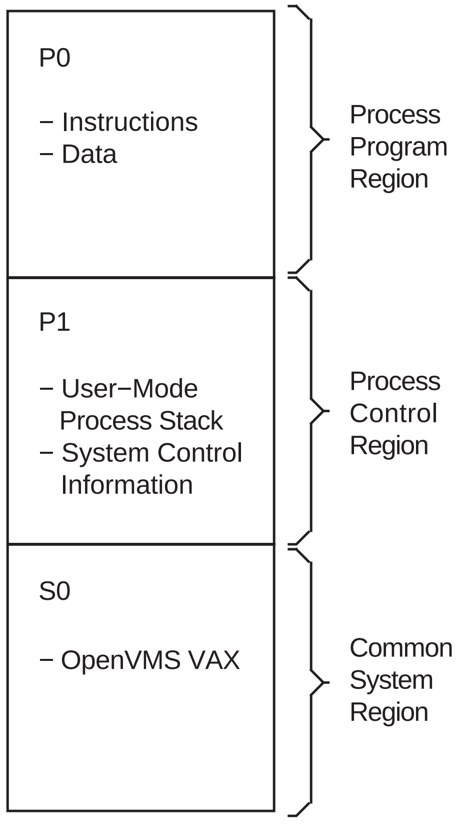
- Common system region (S0)

  The common system region is also referred to as S0 space. S0 space contains the operating system. Your program cannot allocate or free memory within the common system region from the user access mode.

  This common system region (S0) of system virtual address space can have appended to it additional virtual address space, known as a reserved region, or S1 space, that creates a single region of system space. As a result, the system virtual address space increases from 1 GB to 2 GB.

A summary of these regions appears in Figure 19–1.

Figure 19–1 Virtual Address Overview on VAX Systems



ZK−4145−GE

The memory management routines map and control the relationship between physical memory and the virtual address space of a process. These activities are, for the most part, transparent to you and your programs. In some cases, however, you can make a program more efficient by explicitly controlling its virtual memory usage.

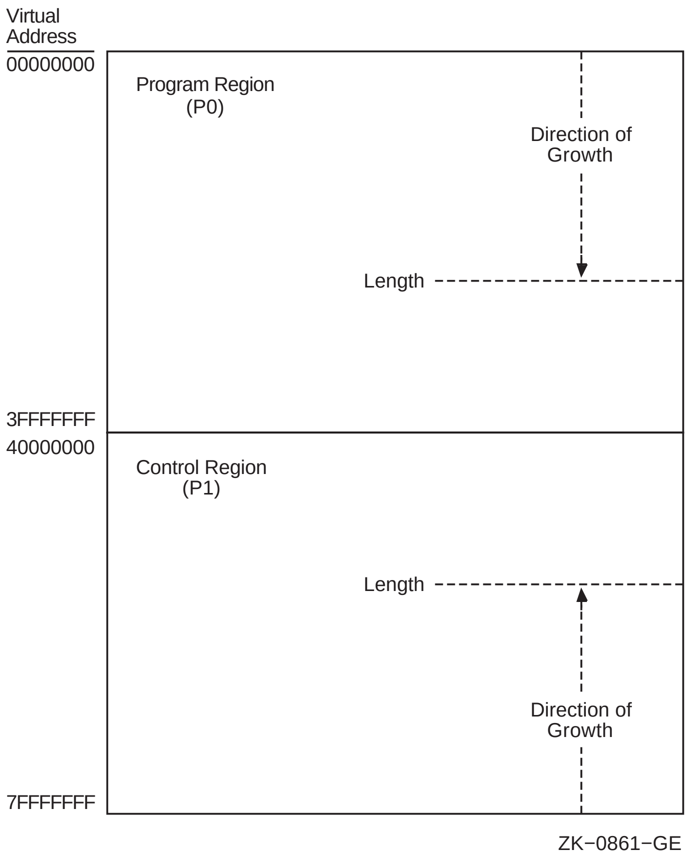
The maximum size to which a process can increase its address space is controlled by the system parameter VIRTUALPAGECNT.

Using memory management system services, a process can add a specified number of pages to the end of either the program region or the control region. Adding pages to the program region provides the process with additional space for image execution, for example, for the dynamic creation of tables or data areas. Adding pages to the control region increases the size of the user stack.

As new pages are referenced, the stack is automatically expanded, as shown in Figure 19–2. (By using the STACK= option in a linker options file, you can also expand the user stack when you link the image.)

Figure 19–2 illustrates the layout of a process's virtual memory. The initial size of a process's virtual address space depends on the size of the image being executed.

Figure 19–2 Layout of VAX Process Virtual Address Space



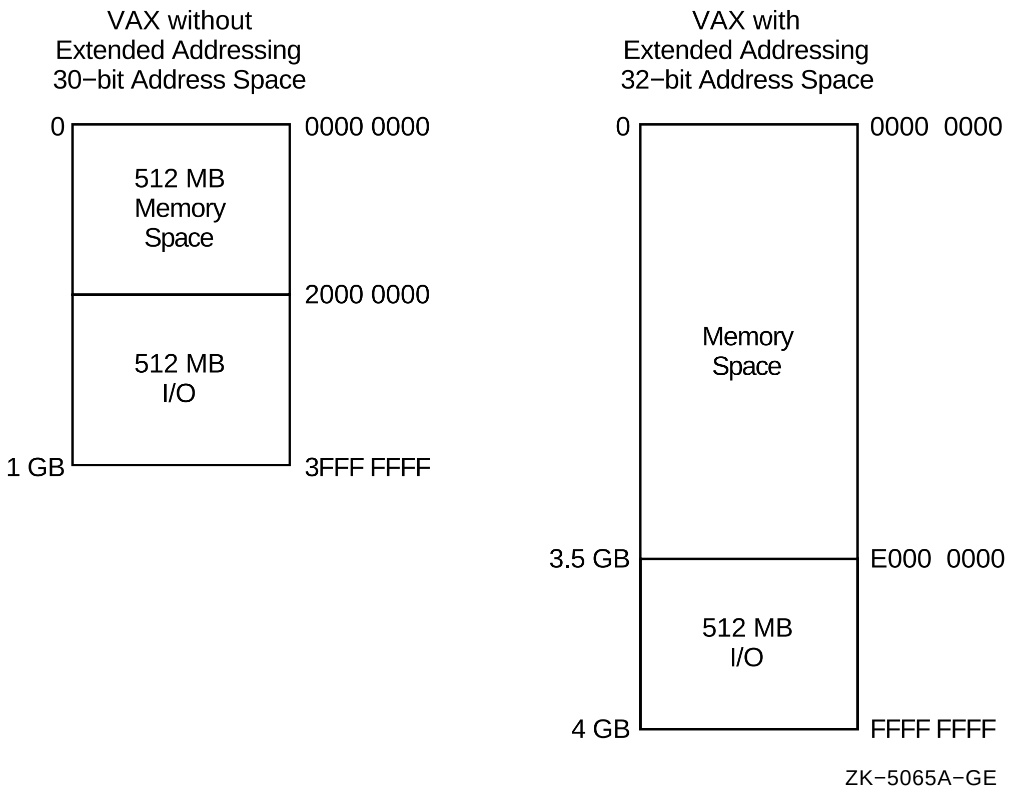
19.3 Extended Addressing Enhancements on Selected VAX Systems

Selected VAX systems have extended addressing (XA) as part of the memory management subsystem. Extended addressing enhancement is supported on the VAX 6000 Model 600, VAX 7000 Model600, and VAX 10000 Model 600 systems. Extended addressing contains the following two major enhancements that affect system images, system integrated products (SIPs), privileged layered products (LPs), and device drivers:

- Extended physical addressing (XPA)
- Extended virtual addressing (XVA)

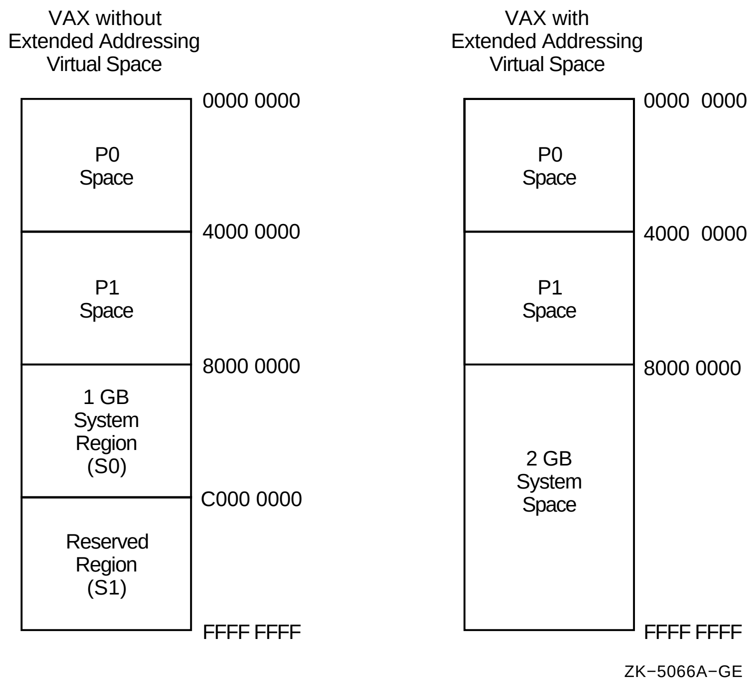
Extended physical addressing increases the size of a physical address from 30 bits to 32 bits. This increases the capacity for physical memory from 512 MB to 3.5 GB as shown in Figure 19–3. The 512 MB is still reserved for I/O and adapter space.

Figure 19–3 Physical Address Space for VAX Systems with XPA



Extended virtual addressing (XVA) increases the size of the virtual page number field in the format of a system space address from 21 bits to 22 bits. The region of system virtual address space, known as the reserved region or S1 space, is appended to the existing region of system virtual address space known as S0 space, thereby creating a single region of system space. As a result, the system virtual address space increases from 1 GB to 2 GB as shown in Figure 19–4.

Figure 19–4 Virtual Address Space for VAX Systems with XVA



19.3.1 Page Table Entry for Extended Addresses on VAX Systems

As shown in Figure 19–3, extended addressing increases the maximum physical address space supported by VAX systems from 1 GB to 4 GB. This is accomplished by expanding the page frame number (PFN) field in a page table entry (PTE) from 21 bits to 23 bits, and implementing changes in the memory management arrays that are indexed by PFN. Both the process page table entry and system page table entry are changed.

19.4 Levels of Memory Allocation Routines

Sophisticated software systems must often create and manage complex data structures. In these systems, the size and number of elements are not always known in advance. You can tailor the memory allocation for these elements by using **dynamic memory allocation**. By managing the memory allocation, you can avoid allocating fixed tables that may be too large or too small for your program. Managing memory directly can improve program efficiency. By allowing you to allocate specific amounts of memory, the operating system provides a hierarchy of routines and services for memory management. Memory allocation and deallocation routines allow you to allocate and free storage within the virtual address space available to your process.

There are three levels of memory allocation routines:

1. Memory management system services

   The memory management system services comprise the lowest level of memory allocation routines. These services include, but are not limited to, the following:

   SYS$EXPREG (Expand Region)
   SYS$CRETVA (Create Virtual Address Space)
   SYS$DELTVA (Delete Virtual Address Space)

SYS$CRMPSC (Create and Map Section)
SYS$MGBLSC (Map Global Section)
SYS$DGBLSC (Delete Global Section)

For most cases in which a system service is used for memory allocation, the Expand Region (SYS$EXPREG) system service is used to create pages of virtual memory.

Because system services provide more control over allocation procedures than RTL routines, you must manage the allocation precisely. System services provide extensive control over address space allocation by allowing you to do the following types of tasks:

- Add or delete virtual address space to the process program region (P0) or process control region (P1)
- Add or delete virtual address space at a specific range of addresses
- Increase or decrease the number of pages in a program's working set
- Lock or delete pages of a program's working set in memory
- Lock the entire program's working set in memory (by disabling process swapping)
- Define disk files containing data or shareable images and map the files into the virtual address space of a process

2. RTL page management routines

   RTL routines are available for creating, deleting, and accessing information about virtual address space. You can either allocate a specified number of contiguous pages or create a zone of virtual address space. A **zone** is a logical unit of the memory pool or subheap that you can control as an independent area. It can be any size required by your program. Refer to Chapter 21 for more information about zones.

   The RTL page management routines LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE provide a convenient mechanism for allocating and freeing pages of memory.

   These routines maintain a processwide pool of free pages. If unallocated pages are not available when LIB$GET_VM_PAGE is called, it calls SYS$EXPREG to create the required pages in the program region (P0 space).

3. RTL heap management routines

   The RTL heap management routines LIB$GET_VM and LIB$FREE_VM provide a mechanism for allocating and freeing blocks of memory of arbitrary size.

   The following are heap management routines based on the concept of zones:

   LIB$CREATE_VM_ZONE
   LIB$CREATE_USER_VM_ZONE
   LIB$DELETE_VM_ZONE
   LIB$FIND_VM_ZONE
   LIB$RESET_VM_ZONE
   LIB$SHOW_VM_ZONE
   LIB$VERIFY_VM_ZONE

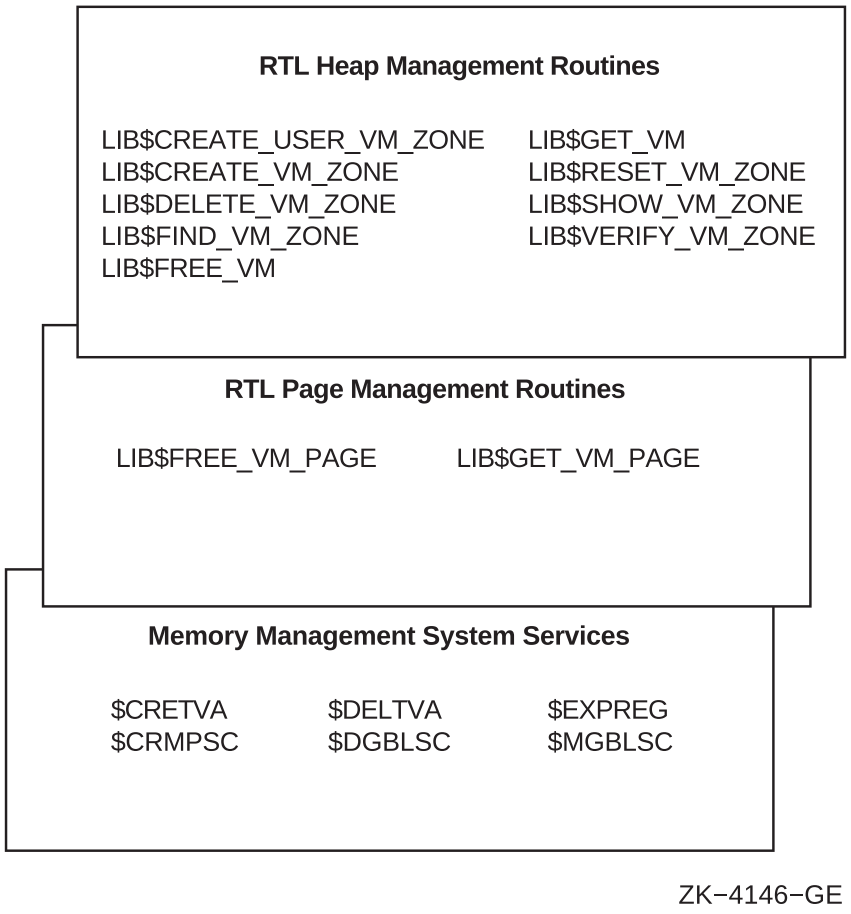
   If an unallocated block is not available to satisfy a call to LIBGET_VM, LIBGET_VM calls LIB$GET_VM_PAGE to allocate additional pages.

Modular application programs can call routines at any or all levels of the hierarchy, depending on the kinds of services the application program needs. You must observe the following basic rule when using multiple levels of the hierarchy:

> Memory that is allocated by an allocation routine at one level of the hierarchy must be freed by calling a deallocation routine at the same level of the hierarchy. For example, if you allocated a page of memory by calling LIB$GET_VM_PAGE, you can free it only by calling LIB$FREE_VM_PAGE.

Figure 19–5 shows the three levels of memory allocation routines.

Figure 19–5 Hierarchy of VAX Memory Management Routines



For information about using memory management RTLs, see Chapter 21.

19.5 Using System Services for Memory Allocation

This section describes how to use system services to perform the following tasks:

- Increase and decrease virtual address space
- Input and return address arrays
- Control page ownership and protection
- Control working set paging
- Control process swapping

19.5.1 Increasing and Decreasing Virtual Address Space

The system services allow you to add address space anywhere within the process's program region (P0) or control region (P1). To add address space at the end of P0 or P1, use the Expand Program/Control Region (SYS$EXPREG) system service. SYS$EXPREG optionally returns the range of virtual addresses for the new pages. To add address space in other portions of P0 or P1, use SYS$CRETVA.

The format for SYS$EXPREG is as follows:

SYS$EXPREG (pagcnt,[retadr],[acmode],[region])

Specifying the Number of Pages

Use the **pagcnt** argument to specify the number of pages to add to the end of the region. The range of addresses where the new pages are added is returned in **retadr**.

Specifying the Access Mode

Use the **acmode** argument to specify the access to be assigned to the newly created pages.

Specifying the Region

Use the **region** argument to specify whether to add the pages to the end of the P0 or P1 region. This argument is optional.

To deallocate pages allocated with SYS$EXPREG, use SYS$DELTVA.

The following example illustrates how to add 4 pages to the program region of a process by writing a call to the SYS$EXPREG system service:

```
#include <stdio.h>
#include <ssdef.h>

main() {
    unsigned int status, retadr[1],pagcnt=4, region=0;

/* Add 4 pages to P0 space */
    status = SYS$EXPREG( pagcnt, &retadr, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d Ending address: %d\n",
               retadr.lower,retadr.upper);
}
```

The value 0 is passed in the **region** argument to specify that the pages are to be added to the program region. To add the same number of pages to the control region, you would specify REGION=#1.

Note that the **region** argument to SYS$EXPREG is optional; if it is not specified, the pages are added to or deleted from the program region by default.

The SYS$EXPREG service can add pages only to the end of a particular region. When you need to add pages to the middle of these regions, you can use the Create Virtual Address Space (SYS$CRETVA) system service. Likewise, when you need to delete pages created by either SYS$EXPREG or SYS$CRETVA, you can use the Delete Virtual Address Space (SYS$DELTVA) system service. For example, if you have used the SYS$EXPREG service twice to add pages to the program region and want to delete the first range of pages but not the second, you could use the SYS$DELTVA system service, as shown in the following example:

```
#include <stdio.h>
#include <ssdef.h>

struct {
    unsigned int lower, upper;
}retadr1, retadr2, retadr3;

main() {
    unsigned int status, pagcnt=4, region=0;

/* Add 4 pages to P0 space */
    status = SYS$EXPREG( pagcnt, &retadr1, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr1.lower,retadr1.upper);

/* Add 3 more pages to P0 space */

    pagcnt = 3;
    status = SYS$EXPREG( pagcnt, &retadr2, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr2.lower,retadr2.upper);

/* Delete original allocation */
    status = SYS$DELTVA( &retadr1, &retadr3, 0 );
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr1.lower,retadr1.upper);

}
```

In this example, the first call to SYS$EXPREG adds 4 pages to the program region; the virtual addresses of the created pages are returned in the 2-longword array at retadr1. The second call adds 3 pages and returns the addresses at retadr2. The call to SYS$DELTVA deletes the first 4 pages that were added.

Caution

Be aware that using SYS$CRETVA presents some risk because it can delete pages that already exist if those pages are not owned by a more privileged access mode. Further, if those pages are deleted, no notification is sent. Therefore, unless you have complete control over an entire system, use SYS$EXPREG or the RTL routines to allocate address space.

19.5.2 Input Address Arrays and Return Address Arrays

When the SYS$EXPREG system service adds pages to a region, it adds them in the normal direction of growth for the region. The return address array, if requested, indicates the order in which the pages were added. For example:

- If the program region is expanded, the starting virtual address is smaller than the ending virtual address.
- If the control region is expanded, the starting virtual address is larger than the ending virtual address.

The addresses returned indicate the first byte in the first page that was added or deleted and the last byte in the last page that was added or deleted.

When input address arrays are specified for the Create and Delete Virtual Address Space (SYS$CRETVA and SYS$DELTVA, respectively) system services, these services add or delete pages beginning with the address specified in the first longword and ending with the address specified in the second longword.

Note

The operating system always adjusts the starting and ending virtual addresses up or down to fit page boundaries.

The order in which the pages are added or deleted does not have to be in the normal direction of growth for the region. Moreover, because these services add or delete only whole pages, they ignore the low-order 9 bits of the specified virtual address (the low-order 9 bits contain the byte offset within the page). The virtual addresses returned indicate the byte offsets.

Table 19–1 shows some sample virtual addresses in hexadecimal that may be specified as input to SYS$CRETVA or SYS$DELTVA and shows the return address arrays if all pages are successfully added or deleted.

Table 19–1 Sample Virtual Address Arrays on VAX Systems

Input Array		Output Array			
Start	**End**	**Region**	**Start**	**End**	**Number of Pages**
1010	1670	P0	1000	17FF	4
1450	1451	P0	1400	15FF	1
1200	1000	P0	1000	13FF	2
1450	1450	P0	1400	15FF	1
7FFEC010	7FFEC010	P1	7FFEC1FF	7FFEC000	1
7FFEC010	7FFEBCA0	P1	7FFEC1FF	7FFEBC00	3

Note that if the input virtual addresses are the same, as in the fourth and fifth items in Table 19–1, a single page is added or deleted. The return address array indicates that the page was added or deleted in the normal direction of growth for the region.

19.5.3 Page Ownership and Protection

Each page in the virtual address space of a process is owned by the access mode that created the page. For example, pages in the program region that are initially provided for the execution of an image are owned by user mode. Pages that the image creates dynamically are also owned by user mode. Pages in the control region, except for the pages containing the user stack, are normally owned by more privileged access modes.

Only the owner access mode or a more privileged access mode can delete the page or otherwise affect it. The owner of a page can also indicate, by means of a protection code, the type of access that each access mode will be allowed.

The Set Protection on Pages (SYS$SETPRT) system service changes the protection assigned to a page or group of pages. The protection is expressed as a code that indicates the specific type of access (none, read-only, read/write) for each of the four access modes (kernel, executive, supervisor, user). Only the owner access mode or a more privileged access mode can change the protection for a page.

When an image attempts to access a page that is protected against the access attempted, a hardware exception called an **access violation** occurs. When an image calls a system service, the service probes the pages to be used to determine whether an access violation would occur if the image attempts to read or write one of the pages. If an access violation would occur, the service exits with the status code SS$_ACCVIO.

Because the memory management services add, delete, or modify a single page at a time, one or more pages can be successfully changed before an access violation is detected. If the **retadr** argument is specified in the service call, the service returns the addresses of pages changed (added, deleted, or modified) before the error. If no pages are affected, that is, if an access violation would occur on the first page specified, the service returns a value of -1 in both longwords of the return address array.

If the **retadr** argument is not specified, no information is returned.

19.5.4 Working Set Paging

When a process is executing an image, a subset of its pages resides in physical memory; these pages are called the **working set** of the process. The working set includes pages in both the program region and the control region. The initial size of a process's working set is usually defined by the process's working set default (WSDEFAULT) quota. The maximum size of a process's working set is normally defined by the process's working set quota (WSQUOTA). When ample memory is available, a process's working-set upper growth limit can be expanded by its working set extent (WSEXTENT).

When the image refers to a page that is not in memory, a page fault occurs and the page is brought into memory, replacing an existing page in the working set. If the page that is going to be replaced is modified during the execution of the image, that page is written into a paging file on disk. When this page is needed again, it is brought back into memory, again replacing a current page from the working set. This exchange of pages between physical memory and secondary storage is called **paging.**

The paging of a process's working set is transparent to the process. However, if a program is very large or if pages in the program image that are used often are being paged in and out frequently, the overhead required for paging may decrease the program's efficiency. The SYS$ADJWSL, SYS$PURGWS, and SYS$LKWSET system services allow a process, within limits, to counteract these potential problems.

SYS$ADJWSL System Service

The Adjust Working Set Limit (SYS$ADJWSL) system service increases or decreases the maximum number of pages that a process can have in its working set. The format for this routine is as follows:

```
SYS$ADJWSL ([pagcnt],[wsetlm])
```

Use the **pagcnt** argument to specify the number of pages to add or subtract from the current working set size. The new working set size is returned in **wsetlm**.

SYS$PURGWS System Service

The Purge Working Set (SYS$PURGWS) system service removes one or more pages from the working set.

SYS$LKWSET System Service

The Lock Pages in Working Set (SYS$LKWSET) system service makes one or more pages in the working set ineligible for paging by locking them in the working set. Once locked into the working set, those pages remain until they are explicitly unlocked with the Unlock Pages in Working Set (SYS$ULWSET) system service or until program execution ends. The format is as follows:

SYS$LKWSET (inadr,[retadr],[acmode])

Specifying a Range of Addresses Use the **inadr** argument to specify the range of addresses to be locked. The range of addresses of the pages actually locked are returned in the **retadr** argument.

Specifying the Access Mode Use the **acmode** argument to specify the access mode to be associated with the pages you want locked.

19.5.5 Process Swapping

The operating system balances the needs of all the processes currently executing, providing each with the system resources it requires on an as-needed basis. The memory management routines balance the memory requirements of the process. Thus, the sum of the working sets for all processes currently in physical memory is called the **balance set.**

When a process whose working set is in memory becomes inactive—for example, to wait for an I/O request or to hibernate—the entire working set or part of it may be removed from memory to provide space for another process's working set to be brought in for execution. This removal from memory is called **swapping.**

The working set may be removed in two ways:

- Partially—Also called **swapper trimming.** Pages are removed from the working set of the target process so that the number of pages in the working set is fewer, but the working set is not swapped.
- Entirely—Called swapping. All pages are swapped out of memory.

When a process is swapped out of the balance set, all the pages (both modified and unmodified) of its working set are swapped, including any pages that had been locked in the working set.

A privileged process may lock itself in the balance set. While pages can still be paged in and out of the working set, the process remains in memory even when it is inactive. To lock itself in the balance set, the process issues the Set Process Swap Mode (SYS$SETSWM) system service, as follows:

```
$SETSWM_S SWPFLG=#1
```

This call to SYS$SETSWM disables process swap mode. You can also disable swap mode by setting the appropriate bit in the STSFLG argument to the Create Process (SYS$CREPRC) system service; however, you need the PSWAPM privilege to alter process swap mode.

A process can also lock particular pages in memory with the Lock Pages in Memory (SYS$LCKPAG) system service. These pages are not part of the process's working set, but they are forced into the process's working set. When pages are locked in memory with this service, the pages remain in memory even when the remainder of the process's working set is swapped out of the balance set. These remaining pages stay in memory until they are unlocked with SYS$ULKPAG. SYS$LCKPAG can be useful in special circumstances, for example, for routines that perform I/O operations to devices without using the operating system's I/O system.

You need the PSWAPM privilege to issue SYS$LCKPAG or SYS$ULKPAG.

19.5.6 Sections

A **section** is a disk file or a portion of a disk file containing data or instructions that can be brought into memory and made available to a process for manipulation and execution. A section can also be one or more consecutive page frames in physical memory or I/O space; such sections, which require you to specify page frame number (PFN) mapping, are discussed in Section 19.5.6.15.

Sections are either private or global (shared).

- **Private sections** are accessible only by the process that creates them. A process can define a disk data file as a section, map it into its virtual address space, and manipulate it.
- **Global sections** can be shared by more than one process. One copy of the global section resides in physical memory, and each process sharing it refers to the same copy. A global section can contain shareable code or data that can be read, or read and written, by more than one process. Global sections are either temporary or permanent and can be defined for use within a group or on a systemwide basis. Global sections can be either mapped to a disk file or created as a global page-file section.

When modified pages in writable disk file sections are paged out of memory during image execution, they are written back into the section file rather than into the paging file, as is the normal case with files. (However, copy-on-reference sections are not written back into the section file.)

The use of disk file sections involves these two distinct operations:

- The creation of a section defines a disk file as a section and informs the system what portions of the file contain the section.
- The mapping of a section makes it available to a process and establishes the correspondence between virtual blocks in the file and specific addresses in the virtual address space of a process.

The Create and Map Section (SYS$CRMPSC) system service creates and maps a private section or a global section. Because a private section is used only by a single process, creation and mapping are simultaneous operations. In the case of a global section, one process can create a permanent global section and not map to it; other processes can map to it. A process can also create and map a global section in one operation.

The following sections describe the creation, mapping, and use of disk file sections. In each case, operations and requirements that are common to both private sections and global sections are described first, followed by additional notes and requirements for the use of global sections. Section 19.5.6.9 discusses global page-file sections.

19.5.6.1 Creating Sections

To create a disk file section, follow these steps:

1. Open or create the disk file containing the section.
2. Define which virtual blocks in the file comprise the section.
3. Define the characteristics of the section.

19.5.6.2 Opening the Disk File

Before you can use a file as a section, you must open it using OpenVMS Record Management Services (RMS). The following example shows the OpenVMS RMS file access block ($FAB) and $OPEN macros used to open the file and the channel specification to the SYS$CRMPSC system service necessary for reading an existing file:

```
#include <rms.h>
#include <rmsdef.h>
#include <string.h>
#include <secdef.h>

struct FAB secfab;

main() {
    unsigned short chan;
    unsigned int status, retadr[1], pagcnt=1, flags;
    char *fn = "SECTION.TST";

/* Initialize FAB fields */
    secfab = cc$rms_fab;
    secfab.fab$l_fna = fn;
    secfab.fab$b_fns = strlen(fn);
    secfab.fab$l_fop = FAB$M_CIF;
    secfab.fab$b_rtv = -1;

/* Create a file if none exists */
    status = SYS$CREATE( &secfab, 0, 0 );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    flags = SEC$M_EXPREG;
    chan = secfab.fab$l_stv;
    status = SYS$CRMPSC(0, &retadr, 0, 0, 0, 0, flags, chan, pagcnt, 0, 0, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

}
```

In this example, the file options parameter (FOP) indicates that the file is to be opened for user I/O; this parameter is required so that OpenVMS RMS assigns the channel using the access mode of the caller. OpenVMS RMS returns the channel number on which the file is accessed; this channel number is specified as input to SYS$CRMPSC (**chan** argument). The same channel number can be used for multiple create and map section operations.

The option RTV=–1 tells the file system to keep all of the pointers to be mapped in memory at all times. If this option is omitted, SYS$CRMPSC requests the file system to expand the pointer areas, if necessary. Storage for these pointers is charged to the BYTLM quota, which means that opening a badly fragmented file can fail with an EXBYTLM failure status. Too many fragmented sections may cause the byte limit to be exceeded.

The file may be a new file that is to be created while it is in use as a section. In this case, use the $CREATE macro to open the file. If you are creating a new file, the file access block (FAB) for the file must specify an allocation quantity (ALQ parameter).

You can also use SYS$CREATE to open an existing file; if the file does not exist, it is created. The following example shows the required fields in the FAB for the conditional creation of a file:

```
GBLFAB: $FAB    FNM=<GLOBAL.TST>, -
                ALQ=4, -
                FAC=PUT,-
                FOP=<UFO,CIF,CBT>, -
                SHR=<PUT,UPI>
           .
           .
           .
        $CREATE FAB=GBLFAB
```

When the $CREATE macro is invoked, it creates the file GLOBAL.TST if the file does not currently exist. The CBT (contiguous best try) option requests that, if possible, the file be contiguous. Although section files are not required to be contiguous, better performance can result if they are.

19.5.6.3 Defining the Section Extents

After the file is opened successfully, SYS$CRMPSC can create a section either from the entire file or from certain portions of it. The following arguments to SYS$CRMPSC define the extents of the file that comprise the section:

- **pagcnt** (page count). This argument is required. It indicates the number of virtual blocks that will be mapped. These blocks correspond to pages in the section.
- **vbn** (virtual block number). This argument is optional. It defines the number of the virtual block in the file that is the beginning of the section. If you do not specify this argument, the value 1 is passed (the first virtual block in the file is the beginning of the section). If you have specified physical page frame number (PFN) mapping, the **vbn** argument specifies the starting PFN. The system does not allow you to specify a virtual block number outside of the file.

19.5.6.4 Defining the Section Characteristics

The **flags** argument to SYS$CRMPSC defines the following section characteristics:

- Whether it is a private section or a global section. The default is to create a private section.
- How the pages of the section are to be treated when they are copied into physical memory or when a process refers to them. The pages in a section can be either or both of the following:
  - Read/write or read-only
  - Created as demand-zero pages or as copy-on-reference pages, depending on how the processes are going to use the section and whether the file contains any data (see Section 19.5.6.10)
- Whether the section is to be mapped to a disk file or to specific physical page frames (see Section 19.5.6.15).

Table 19–2 shows the flag bits that must be set for specific characteristics.

Table 19–2 Flag Bits to Set for Specific Section Characteristics on VAX Systems

	Section to Be Created				
Correct Flag Combinations	**Private**	**Global**	**PFN Private**	**PFN Global**	**Shared Memory**
SEC$M_GBL	0	1	0	1	1
SEC$M_CRF	Optional	Optional	0	0	0
SEC$M_DZRO	Optional	Optional	0	0	Optional
SEC$M_WRT	Optional	Optional	Optional	Optional	Optional
SEC$M_PERM	Not used	Optional	Optional	1	1
SEC$M_SYSGBL	Not used	Optional	Not used	Optional	Optional
SEC$M_PFNMAP	0	0	1	1	0
SEC$M_EXPREG	Optional	Optional	Optional	Optional	Optional
SEC$M_PAGFIL	0	Optional	0	0	0

When you specify section characteristics, the following restrictions apply:

- Global sections cannot be both demand-zero and copy-on-reference.
- Demand-zero sections must be writable.
- Shared memory private sections are not allowed.

19.5.6.5 Defining Global Section Characteristics

If the section is a global section, you must assign it a character string name (**gsdnam** argument) to it so that other processes can identify it when they map it. The format of this character string name is explained in Section 19.5.6.6.

The **flags** argument specifies the following types of global sections:

- Group temporary (the default)
- Group permanent
- System temporary
- System permanent

Group global sections can be shared only by processes executing with the same group number. The name of a group global section is implicitly qualified by the group number of the process that created it. When other processes map it, their group numbers must match.

A temporary global section is automatically deleted when no processes are mapped to it, but a permanent global section remains in existence even when no processes are mapped to it. A permanent global section must be explicitly marked for deletion with the Delete Global Section (SYS$DGBLSC) system service.

You need the user privileges PRMGBL and SYSGBL to create permanent group global sections or system global sections (temporary or permanent), respectively.

A system global section is available to all processes in the system.

Optionally, a process creating a global section can specify a protection mask (**prot** argument), restricting all access or a type of access (read, write, execute, delete) to other processes.

19.5.6.6 Global Section Name

The **gsdnam** argument specifies a descriptor that points to a character string.

Translation of the **gsdnam** argument proceeds in the following manner:

1. The current name string is prefixed with GBL$ and the result is subject to logical name translation.
2. If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the system parameter LNM$C_MAXDEPTH.
3. The GBL$ prefix is stripped from the current name string that could not be translated. This current string is the name of the global section.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE GBL$GSDATA GSDATA_001
```

Your program contains the following statements:

```
#include <descrip.h>
   .
   .
   .
    $DESCRIPTOR(gsdnam,"GSDATA");
   .
   .
   .
    status = sys$crmpsc(&gsdnam,  . . .  );
```

The following logical name translation takes place:

1. GBL$ is prefixed to GDSDATA.
2. GBL$GSDATA is translated to GSDATA_001. (Further translation is not successful. When logical name translation fails, the string is passed to the service.)

There are three exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), the operating system strips the underscore and considers the resultant string to be the actual name (that is, further translation is not performed).
- If the name string is the result of a logical name translation, then the name string is checked to see whether it has the **terminal** attribute. If the name string is marked with the **terminal** attribute, the operating system considers the resultant string to be the actual name (that is, further translation is not performed).
- If the global section has a name in the format *name_nnn*, the operating system first strips the underscore and the digits *(nnn)*, then translates the resultant name according to the sequence discussed in this section, and finally reappends the underscore and digits. The system uses this method

in conjunction with known images and shared files installed by the system manager.

19.5.6.7 Mapping Sections

When you call SYS$CRMPSC to create or map a section, or both, you must provide the service with a range of virtual addresses (**inadr** argument) into which the section is to be mapped.

If you know specifically which pages the section should be mapped into, you provide these addresses in a 2-longword array. For example, to map a private section of 10 pages into virtual pages 10 through 19 of the program region, specify the input address array as follows:

```
unsigned int maprange[1];

maprange[0]= 0x1400; /* Address (hex) of page 10 */
maprange[1]= 0x2300; /* Address (hex) of page 19 */
```

You do not need to know the explicit addresses to provide an input address range. If you want the section mapped into the first available virtual address range in the program region (P0) or the control region (P1), you can specify the SEC$M_EXPREG flag bit in the **flags** argument. In this case, the addresses specified by the **inadr** argument control whether the service finds the first available space in P0 or P1. The value specified or defaulted for the **pagcnt** argument determines the number of pages mapped. The following example shows part of a program used to map a section at the current end of the program region:

```
    unsigned int status, inadr[1], retadr[1], flags;

    inadr[0]= 0x200; /* Any program (P0) region address */
    inadr[1]= 0x200; /* Any P0 address (can be same) */

   .
   .
   .
/* Address range returned in retadr */

    flags = SEC$M_EXPREG;
    status = sys$crmpsc(&inadr, &retadr, flags,  . . .  );
```

The addresses specified do not have to be currently in the virtual address space of the process. SYS$CRMPSC creates the required virtual address space during the mapping of the section. If you specify the **retadr** argument, the service returns the range of addresses actually mapped.

After a section is mapped successfully, the image can refer to the pages using one of the following:

- A base register or pointer and predefined symbolic offset names
- Labels defining offsets of an absolute program section or structure

The following example shows part of a program used to create and map a process section:

```
#include <rms.h>
#include <rmsdef.h>
#include <string.h>
#include <secdef.h>

struct FAB secfab;
```

```
main() {
    unsigned short chan;
    unsigned int status, inadr[1], retadr[1], pagcnt=1, flags;
    char *fn = "SECTION.TST";

/* Initialize FAB fields */

    secfab = cc$rms_fab;
    secfab.fab$b_fac = FAB$M_PUT;
    secfab.fab$b_shr = FAB$M_SHRGET || FAB$V_SHRPUT || FAB$V_UPI;
    secfab.fab$l_fna = fn;
    secfab.fab$b_fns = strlen(fn);
    secfab.fab$l_fop = FAB$V_CIF;
    secfab.fab$b_rtv = -1;

/* Create a file if none exists */

    status = SYS$CREATE( &secfab, 0, 0 );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    inadr[0] = X1400;
    inadr[1] = X2300;
    flags = SEC$M_WRT;
    chan = secfab.fab$l_stv;
    status = SYS$CRMPSC(&inadr, &retadr, 0, 0, 0, 0, flags, chan, pagcnt, 0, 0, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

}
```

Notes on Example

1. The OPEN macro opens the section file defined in the file access block SECFAB. (The FOP parameter to the $FAB macro must specify the UFO option.)
2. SYS$CRMPSC uses the addresses specified at MAPRANGE to specify an input range of addresses into which the section will be mapped. The **pagcnt** argument requests that only 4 pages of the file be mapped.
3. The **flags** argument requests that the pages in the section have read/write access. The symbolic flag definitions for this argument are defined in the $SECDEF macro. Note that the file access field (FAC parameter) in the FAB also indicates that the file is to be opened for writing.
4. When SYS$CRMPSC completes, the addresses of the 4 pages that were mapped are returned in the output address array at RETRANGE. The address of the beginning of the section is placed in general register 6, which serves as a pointer to the section.

19.5.6.8 Mapping Global Sections

A process that creates a global section can map that global section. Then other processes can map it by calling the Map Global Section (SYS$MGBLSC) system service.

When a process maps a global section, it must specify the global section name assigned to the section when it was created, whether it is a group or system global section, and whether it wants read-only or read/write access. The process may also specify the following:

- A version identification (**ident** argument), indicating the version number of the global section (when multiple versions exist) and whether more recent versions are acceptable to the process.

- A relative page number (**relpag** argument) that specifies the page number relative to the beginning of the section to begin mapping the section. In this way, processes can use only portions of a section. Additionally, a process can map a piece of a section into a particular address range and subsequently map a different piece of the section into the same virtual address range.

To specify that the global section being mapped is located in physical memory that is being shared by multiple processors, you can include the shared memory name in the **gsdnam** argument character string (see Section 19.5.6.6). A demand-zero global section in memory shared by multiple processors must be mapped when it is created.

Cooperating processes can issue a call to SYS$CRMPSC to create and map the same global section. The first process to call the service actually creates the global section; subsequent attempts to create and map the section result only in mapping the section for the caller. The successful return status code SS$_CREATED indicates that the section did not already exist when the SYS$CRMPSC system service was called. If the section did exist, the status code SS$_NORMAL is returned.

The example in Section 19.5.6.10 shows one process (ORION) creating a global section and a second process (CYGNUS) mapping the section.

19.5.6.9 Global Page-File Sections

Global page-file sections are used to store temporary data in a global section. A global page-file section is a section of virtual memory that is not mapped to a file. The section can be deleted when processes have finished with it. (Contrast this to demand-zero pages, where initialization is not necessary but the pages are saved in a file.) The system parameter GBLPAGFIL controls the total number of global page-file pages in the system.

To create a global page-file section, you must set the flag bits SEC$M_GBL and SEC$M_PAGFIL in the **flags** argument to the Create and Map Section (SYS$CRMPSC) system service. The channel (**chan** argument) must be 0.

You cannot specify the flag bit SEC$M_CRF with the flag bit SEC$M_PAGFIL.

19.5.6.10 Section Paging

The first time an image executing in a process refers to a page that was created during the mapping of a disk file section, the page is copied into physical memory. The address of the page in the virtual address space of a process is mapped to the physical page. During the execution of the image, normal paging can occur; however, pages in sections are not written into the page file when they are paged out, as is the normal case. Rather, if they have been modified, they are written back into the section file on disk. The next time a page fault occurs for the page, the page is brought back from the section file.

If the pages in a section were defined as demand-zero pages or copy-on-reference pages when the section was created, the pages are treated differently, as follows:

- If the call to SYS$CRMPSC requested that pages in the section be treated as demand-zero pages, these pages are initialized to zero when they are created in physical memory. If the file is either a new file being created as a section or a file being completely rewritten, demand-zero pages provide a convenient way of initializing the pages. The pages are paged back into the section file.
- When the virtual address space is deleted, all unreferenced pages are written back to the file as zeros. This causes the file to be initialized, no matter how few pages were modified.

- If the call to SYS$CRMPSC requested that pages in the section be copy-on-reference pages, each process that maps to the section receives its own copy of the section, on a page-by-page basis from the file, as it refers to them. These pages are never written back into the section file but are paged to the paging file as needed.

In the case of global sections, more than one process can be mapped to the same physical pages. If these pages need to be paged out or written back to the disk file defined as the section, these operations are done only when the pages are not in the working set of any process.

In the following example, process ORION creates a global section, and process CYGNUS maps to that section:

```
/* Process ORION */

#include <rms.h>
#include <rmsdef.h>
#include <string.h>
#include <secdef.h>
#include <descrip.h>

struct FAB gblfab;

main() {
    unsigned short chan;
    unsigned int status, flags, efn=65;
    char *fn = "SECTION.TST";
    $DESCRIPTOR(name, "FLAG_CLUSTER");     /* Common event flag cluster name */
    $DESCRIPTOR(gsdnam, "GLOBAL_SECTION"); /* Global section name */

1 status = SYS$ASCEFC(efn, &name, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

/* Initialize FAB fields */

    gblfab = cc$rms_fab;
    gblfab.fab$l_alq = 4;
    gblfab.fab$b_fac = FAB$M_PUT;
    gblfab.fab$l_fnm = fn;
    gblfab.fab$l_fop = FAB$M_CIF | FABM$_CBT;

   .
   .
   .

/* Create a file if none exists */

2 status = SYS$CREATE( &gblfab, 0, 0 );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    flags = SEC$M_GBL || SEC$M_WRT;
    status = SYS$CRMPSC(0, 0, 0, flags, &gsdnam,  . . .  );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$SETEF(efn);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );
   .
   .
   .
}
```

```
/* Process CYGNUS */

    unsigned int status, efn=65;
    $DESCRIPTOR(cluster,"FLAG_CLUSTER");
    $DESCRIPTOR(section,"GLOBAL_SECTION");
   .
   .
   .
3 status = SYS$ASCEFC(efn, &cluster, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$WAITFR(efn);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$MGBLSC(&inadr, &retadr, 0, flags, &section, 0, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

}
```

1 The processes ORION and CYGNUS are in the same group. Each process first associates with a common event flag cluster named FLAG_CLUSTER to use common event flags to synchronize its use of the section.

2 The process ORION creates the global section named GLOBAL_SECTION, specifying section flags that indicate that it is a global section (SEC$M_GBL) and has read/write access. Input and output address arrays, the page count parameter, and the channel number arguments are not shown; procedures for specifying them are the same, as shown in this example.

3 The process CYGNUS associates with the common event flag cluster and waits for the flag defined as FLGSET; ORION sets this flag when it has finished creating the section. To map the section, CYGNUS specifies the input and output address arrays, the flag indicating that it is a global section, and the global section name. The number of pages mapped is the same as that specified by the creator of the section.

19.5.6.11 Reading and Writing Data Sections

Read/write sections provide a way for a process or cooperating processes to share data files in virtual memory.

The sharing of global sections may involve application-dependent synchronization techniques. For example, one process can create and map to a global section in read/write fashion; other processes can map to it in read-only fashion and interpret data written by the first process. Alternatively, two or more processes can write to the section concurrently. (In this case, the application must provide the necessary synchronization and protection.)

After a file is updated, the process or processes can release (or unmap) the section. The modified pages are then written back into the disk file defined as a section.

When this is done, the revision number of the file is incremented, and the version number of the file remains unchanged. A full directory listing indicates the revision number of the file and the date and time that the file was last updated.

19.5.6.12 Releasing and Deleting Sections

A process unmaps a section by deleting the virtual addresses in its own virtual address space to which it has mapped the section. If a return address range was specified to receive the virtual addresses of the mapped pages, this address range can be used as input to the Delete Virtual Address Space (SYS$DELTVA) system service, as follows:

```
$DELTVA_S INADR=RETRANGE
```

When a process unmaps a private section, the section is deleted; that is, all control information maintained by the system is deleted. A temporary global section is deleted when all processes that have mapped to it have unmapped it. Permanent global sections are not deleted until they are specifically marked for deletion with the Delete Global Section (SYS$DGBLSC) system service; they are then deleted when no more processes are mapped.

Note that deleting the pages occupied by a section does not delete the section file but rather cancels the process's association with the file. Moreover, when a process deletes pages mapped to a read/write section and no other processes are mapped to it, all modified pages are written back into the section file.

After a section is deleted, the channel assigned to it can be deassigned. The process that created the section can deassign the channel with the Deassign I/O Channel (SYS$DASSGN) system service, as follows:

```
$DASSGN_S CHAN=GBLFAB+FAB$L_STV
```

19.5.6.13 Writing Back Sections

Because read/write sections are not normally updated on disk until the physical pages they occupy are paged out or until all processes referring to the section have unmapped it, a process should ensure that all modified pages are successfully written back into the section file at regular intervals.

The Update Section File on Disk (SYS$UPDSEC) system service writes the modified pages in a section into the disk file. SYS$UPDSEC is described in the *OpenVMS System Services Reference Manual*.

19.5.6.14 Image Sections

Global sections can contain shareable code. The operating system uses global sections to implement shareable code, as follows:

1. The object module containing code to be shared is linked to produce a shareable image. The shareable image is not, in itself, executable. It contains a series of sections called **image sections.**
2. You link private object modules with the shareable image to produce an executable image. No code or data from the shareable image is put into the executable image.
3. The system manager uses the INSTALL command to create a permanent global section from the shareable image file, making the image sections available for sharing.
4. When you run the executable image, the operating system automatically maps the global sections created by the INSTALL command into the virtual address space of your process.

For details about how to create and identify shareable images and how to link them with private object modules, see the *OpenVMS Linker Utility Manual*. For information about how to install shareable images and make them available for sharing as global sections, see the *OpenVMS System Manager's Manual*.

19.5.6.15 Page Frame Sections

A **page frame section** is one or more contiguous pages of physical memory or I/O space that have been mapped as a section. One use of page frame sections is to map to an I/O page, thus allowing a process to read device registers. A process mapped to an I/O page can also connect to a device interrupt vector.

A page frame section differs from a disk file section in that it is not associated with a particular disk file and is not paged. However, it is similar to a disk file section in most other respects: you create, map, and define the extent and characteristics of a page frame section in essentially the same manner as you do for a disk file section.

To create a page frame section, you must specify page frame number (PFN) mapping by setting the SEC$M_PFNMAP flag bit in the **flags** argument to the Create and Map Section (SYS$CRMPSC) system service. The **vbn** argument is now used to specify that the first page frame is to be mapped instead of the first virtual block. You must have the user privilege PFNMAP to create or delete a page frame section but not to map to an existing one.

Because a page frame section is not associated with a disk file, you do not use the **relpag**, **chan**, and **pfc** arguments to the SYS$CRMPSC service to create or map this type of section. For the same reason, the SEC$M_CRF (copy-on-reference) and SEC$M_DZRO (demand-zero) bit settings in the **flags** argument do not apply. Pages in page frame sections are not written back to any disk file (including the paging file).

Caution

You must use caution when working with page frame sections. If you permit write access to the section, each process that writes to it does so at its own risk. Serious errors can occur if a process writes incorrect data or writes to the wrong page, especially if the page is also mapped by the system or by another process. Thus, any user who has the PFNMAP privilege can damage or violate the security of a system.

19.5.7 Example of Using Memory Management System Services

In the following example, two programs are communicating through a global section. The first program creates and maps a global section (by using SYS$CRMPSC) and then writes a device name to the section. This program also defines the device terminal and process names and sets the event flags that synchronize the processes.

The second program maps the section (by using SYS$MGBLSC) and then reads the device name and the process that allocated the device and any terminal allocated to that process. This program also writes the process named to the terminal global section where the process name can be read by the first program.

The common event cluster is used to synchronize access to the global section. The first program sets REQ_FLAG to indicate that the device name is in the section. The second program sets INFO_FLAG to indicate that the process and terminal names are available.

Data in a section must be page aligned. The following is the option file used at link time that causes the data in the common area named DATA to be page aligned:

PSECT_ATTR = DATA, PAGE

For high-level language usage, use the **solitary** attribute of the linker. See the *OpenVMS Linker Utility Manual* for an explanation of how to use the **solitary** attribute.

Before executing the first program, you need to write a user-open routine that sets the user open bit (FAB$V_UFO) of the FAB options longword (FAB$L_FOP). The user-open routine would then read the channel number that the file is opened on from the status longword (FAB$L_STV) and return that channel number to the main program by using a common block (CHANNEL in this example).

```
!This is the program that creates the global section.

! 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
! (returned from useropen routine)
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 element - first element + size 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))
.
.
.
! This is the program that maps to the global section
! created by the previous program.
! 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
```

20
Memory Management Services and Routines (Alpha Only)

Alpha

This chapter describes the use of memory management system services and run-time routines on Alpha systems. Although the operating system's memory management concepts are much the same on VAX systems and Alpha systems, details of the memory management system are different. These details may be critical to certain uses of the operating system's memory management system services and routines on an Alpha system. This chapter highlights those differences by using the Alpha icon.

This chapter contains the following sections:

Section 20.1 describes the page sizes of Alpha systems.

Section 20.2 describes the layout of virtual address space.

Section 20.3 describes the three levels of the operating system's memory allocation routines.

Section 20.4 discusses how to use system services to add virtual address space, adjust working sets, control process swapping, and create and manage sections. ♦

20.1 Virtual Page Sizes

Alpha

On Alpha systems, in order to facilitate memory protection and mapping, the virtual address space is subdivided into segments of 8 KB, 16 KB, 32 KB, or 64 KB sizes called **CPU-specific pages**. On VAX systems, the page sizes are 512 bytes.

Wherever possible, the Alpha system's versions of the system services and run-time library routines that manipulate memory attempt to preserve compatibility with the VAX system's services and routines. The Alpha system's versions of the routines that accept page count values as arguments still interpret these arguments in 512-byte quantities, which are called **pagelets** to distinguish them from CPU-specific page sizes. The routines convert pagelet values into CPU-specific pages. The routines that return page count values convert from CPU-specific pages to pagelets, so that return values expected by applications are still measured in the same 512-byte units.

This difference in page size does not affect memory allocation using higher-level routines, such as run-time library routines that manipulate virtual memory zones or language-specific memory allocation routines such as the *malloc* and *free* routines in C.

To determine system page size, you make a call to the SYS$GETSYI system service, specifying the SYI$_PAGE_SIZE item code. See the description of SYS$GETSYI and SYI$_PAGE_SIZE in the *OpenVMS System Services Reference Manual* for details. ♦

20.2 Virtual Address Space

The Alpha system defines the same virtual address space layout as the VAX system. The Alpha system virtual address space allows for growth of the P0 and P1 regions in the same directions as on VAX systems.

The initial size of a process's virtual address space depends on the size of the image being executed. The virtual address space of an executing program consists of the following three regions:

- Process program region (P0)

  The process program region is also referred to as P0 space. P0 space contains the instructions and data for the current image.

  Your program can dynamically allocate storage in the process program region by calling run-time library (RTL) dynamic memory allocation routines or the operating system's system services.

- Process control region (P1)

  The process control region is also referred to as P1 space. P1 space contains system control information and the user-mode process stack. The user mode stack expands as necessary toward the lower-addressed end of P1 space.

- Common system region (S0)

  The common system region is also referred to as S0 space, or system space. S0 space contains the operating system. Your program cannot allocate or free memory within the common system region from the user access mode.

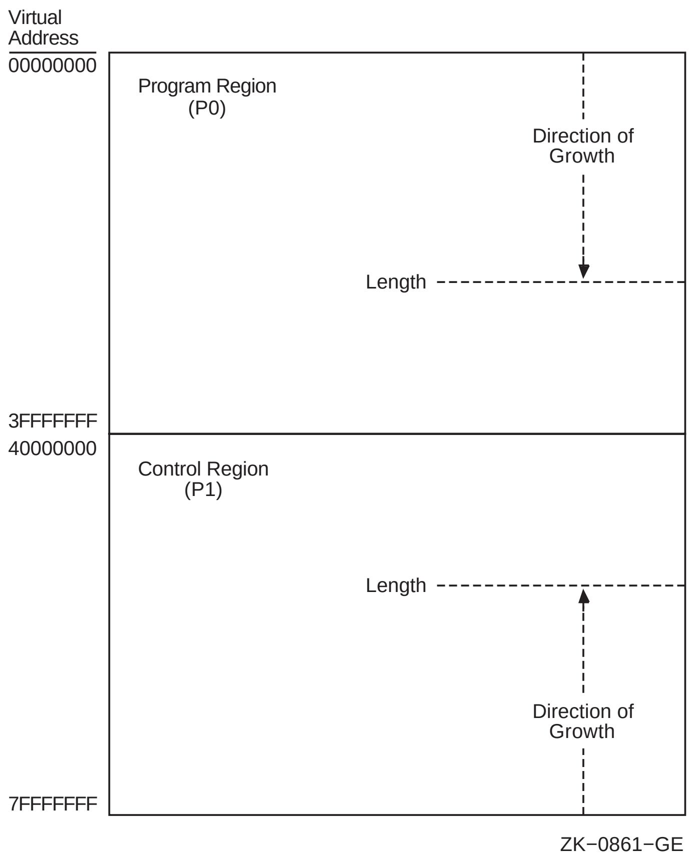
The operating system's memory management routines map and control the relationship between physical memory and the virtual address space of a process. These activities are, for the most part, transparent to you and your programs. In some cases, however, you can make a program more efficient by explicitly controlling its virtual memory usage.

The maximum size to which a process can increase its address space is controlled by the system parameter VIRTUALPAGECNT.

Using memory management system services, a process can add a specified number of pages to the end of either the program region or the control region. Adding pages to the program region provides the process with additional space for image execution, for example, for the dynamic creation of tables or data areas. Adding pages to the control region increases the size of the user stack. As new pages are referenced, the stack is automatically expanded (see Figure 20–1). (By using the STACK= option in a linker options file, you can also expand the user stack when you link the image.)

Figure 20–1 illustrates the layout of a process's virtual memory. The initial size of a process's virtual address space depends on the size of the image being executed.

Figure 20–1 Layout of Alpha Process Virtual Address Space



20.3 Levels of Memory Allocation Routines

Sophisticated software systems must often create and manage complex data structures. In these systems, the size and number of elements are not always known in advance. You can tailor the memory allocation for these elements by using **dynamic memory allocation**. By managing the memory allocation, you can avoid allocating fixed tables that may be too large or too small for your program. Managing memory directly can improve program efficiency. By allowing you to allocate specific amounts of memory, the operating system provides a hierarchy of routines and services for memory management. Memory allocation and deallocation routines allow you to allocate and free storage within the virtual address space available to your process.

There are three levels of memory allocation routines:

1. Memory management system services

   The memory management system services comprise the lowest level of memory allocation routines. These services include, but are not limited to, the following:

   SYS$EXPREG (Expand Region)
   SYS$CRETVA (Create Virtual Address Space)
   SYS$DELTVA (Delete Virtual Address Space)
   SYS$CRMPSC (Create and Map Section)
   SYS$MGBLSC (Map Global Section)

SYS$DGBLSC (Delete Global Section)

For most cases in which a system service is used for memory allocation, the Expand Region (SYS$EXPREG) system service is used to create pages of virtual memory.

Because system services provide more control over allocation procedures than RTL routines, you must manage the allocation precisely. System services provide extensive control over address space allocation by allowing you to do the following types of tasks:

- Add or delete virtual address space to the process's program region (P0) or control region (P1)
- Add or delete virtual address space at a specific range of addresses
- Increase or decrease the number of pages in a program's working set
- Lock or delete pages of a program's working set in memory
- Lock the entire program's working set in memory (by disabling process swapping)
- Define disk files containing data or shareable images and map the files into the virtual address space of a process

2. RTL page management routines

   The RTL routines exist for creating, deleting, and accessing information about virtual address space. You can either allocate a specified number of contiguous pages or create a zone of virtual address space. A **zone** is a logical unit of the memory pool or subheap that you can control as an independent area. It can be any size required by your program. Refer to Chapter 21 for more information about zones.

   The RTL page management routines LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE provide a convenient mechanism for allocating and freeing pages of memory.

   These routines maintain a processwide pool of free pages. If unallocated pages are not available when LIB$GET_VM_PAGE is called, it calls SYS$EXPREG to create the required pages in the program region (P0 space).

3. RTL heap management routines

   The RTL heap management routines LIB$GET_VM and LIB$FREE_VM provide a mechanism for allocating and freeing blocks of memory of arbitrary size.

   The following are heap management routines based on the concept of zones:

   LIB$CREATE_VM_ZONE
   LIB$CREATE_USER_VM_ZONE
   LIB$DELETE_VM_ZONE
   LIB$FIND_VM_ZONE
   LIB$RESET_VM_ZONE
   LIB$SHOW_VM_ZONE
   LIB$VERIFY_VM_ZONE

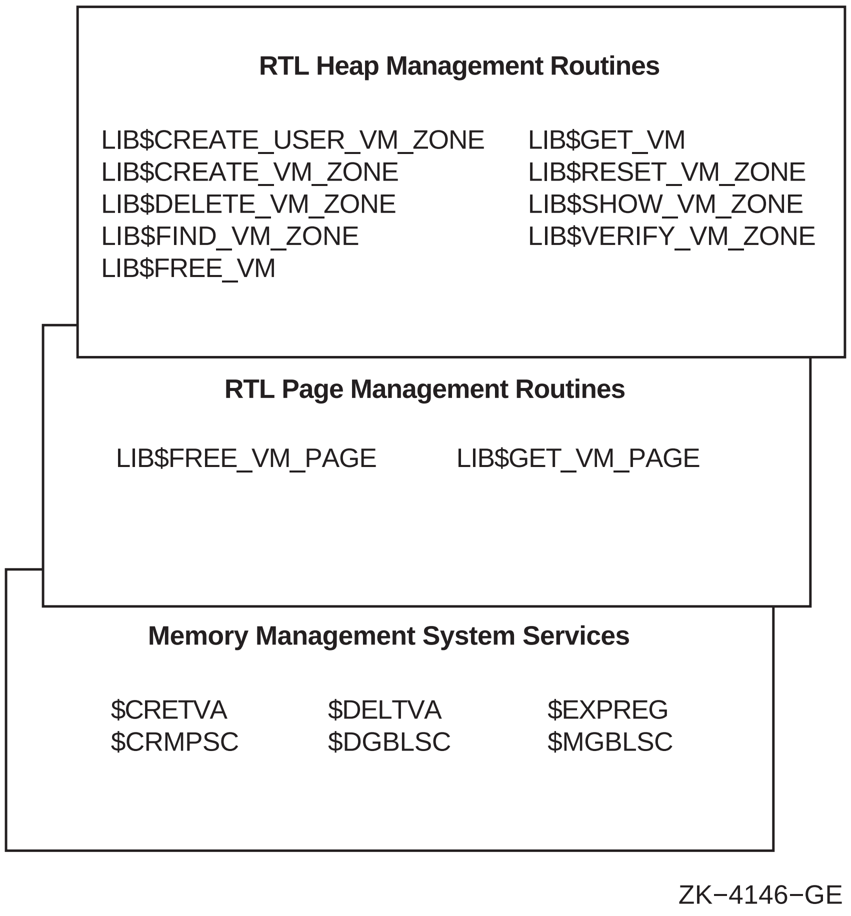
   If an unallocated block is not available to satisfy a call to LIBGET_VM, LIBGET_VM calls LIB$GET_VM_PAGE to allocate additional pages.

Modular application programs can call routines in any or all levels of the hierarchy, depending on the kinds of services the application program needs. You must observe the following basic rule when using multiple levels of the hierarchy:

> Memory that is allocated by an allocation routine at one level of the hierarchy must be freed by calling a deallocation routine at the same level of the hierarchy. For example, if you allocated a page of memory by calling LIB$GET_VM_PAGE, you can free it only by calling LIB$FREE_VM_PAGE.

Figure 20–2 shows the three levels of memory allocation routines.

Figure 20–2 Hierarchy of Alpha Memory Management Routines



For information about using memory management RTLs, see Chapter 21.

20.4 Using System Services for Memory Allocation

This section describes how to use system services to perform the following tasks:

- Increase and decrease virtual address space
- Input and return address arrays
- Control page ownership and protection
- Control working set paging
- Control process swapping

20.4.1 Increasing and Decreasing Virtual Address Space

The system services allow you to add address space anywhere within the process's program region (P0) or control region (P1). To add address space at the end of P0 or P1, use the Expand Program/Control Region (SYS$EXPREG) system service. SYS$EXPREG optionally returns the range of virtual addresses for the new pages. To add address space in other portions of P0 or P1, use SYS$CRETVA.

The format for SYS$EXPREG is as follows:

SYS$EXPREG (pagcnt,[retadr],[acmode],[region])

Specifying the Number of Pages

Alpha

On Alpha systems, use the **pagcnt** argument to specify the number of pagelets to add to the end of the region. The Alpha system rounds the specified pagelet value to the next integral number of Alpha pages for the system where it is executing. To check the exact boundaries of the memory allocated by the system, specify the optional **retadr** argument. The **retadr** argument contains the start address and the end address of the memory allocated by the system service. ♦

Specifying the Access Mode

Use the **acmode** argument to specify the access to be assigned to the newly created pages.

Specifying the Region

Use the **region** argument to specify whether to add the pages to the end of the P0 or P1 region.

To deallocate pages allocated with SYS$EXPREG and SYS$CRETVA, use SYS$DELTVA.

Alpha

For Alpha systems, the following example illustrates the addition of 4 pagelets to the program region of a process by writing a call to the SYS$EXPREG system service.

```
#include <stdio.h>
#include <ssdef.h>

main() {
    unsigned int status, retadr[2],pagcnt=4, region=0;

/* Add 4 pages to P0 space */
    status = SYS$EXPREG( pagcnt, &retadr, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d Ending address: %d\n",
               retadr[0],retadr[1];
}
```

The value 0 is passed in the **region** argument to specify that the pages are to be added to the program region. To add the same number of pages to the control region, you would specify REGION=1.

Note that the **region** argument to the SYS$EXPREG service is optional; if it is not specified, the pages are added to or deleted from the program region by default.♦

Alpha

The SYS$EXPREG service can add pagelets only in the direction of growth of a particular region. When you need to add pages to the middle of these regions, you can use the Create Virtual Address Space (SYS$CRETVA) system service. Likewise, when you need to delete pages created by either SYS$EXPREG or SYS$CRETVA, you can use the Delete Virtual Address Space (SYS$DELTVA) system service. For example, if you have used the SYS$EXPREG service twice to add pages to the program region and want to delete the first range of pages but not the second, you could use the SYS$DELTVA system service, as shown in the following example:

```
#include <stdio.h>
#include <ssdef.h>

struct {
    unsigned int lower, upper;
}retadr1, retadr2, retadr3;

main() {
    unsigned int status, pagcnt=4, region=0;

/* Add 4 pages to P0 space */
    status = SYS$EXPREG( pagcnt, &retadr1, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr1.lower,retadr1.upper);

/* Add 3 more pages to P0 space */

    pagcnt = 3;
    status = SYS$EXPREG( pagcnt, &retadr2, 0, region);
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr2.lower,retadr2.upper);

/* Delete original allocation */
    status = SYS$DELTVA( &retadr1, &retadr3, 0 );
    if (( status & 1) != 1)
        LIB$SIGNAL( status );
    else
        printf("Starting address: %d ending address: %d\n",
               retadr1.lower,retadr1.upper);

}
```

In this example, the first call to SYS$EXPREG rounds up the requested pagelet count to an integral number of CPU-specific pages and adds that number of pages to the program region; the virtual addresses of the created pages are returned in the 2-longword array at retadr1. The second request converts the pagelet count to pages, adds them to the program region, and returns the addresses at retadr2. The call to SYS$DELTVA deletes the area created by the first SYS$EXPREG call.

Caution

Be aware that using SYS$CRETVA presents some risk because it can delete pages that already exist if those pages are not owned by a more privileged access mode. Further, if those pages are deleted, notification is not sent. Therefore, unless you have complete control over an entire system, use SYS$EXPREG or the RTL routines to allocate address space.

Section 20.4.3 mentions some other possible risks in using SYS$CRETVA for allocating memory. ♦

20.4.2 Input Address Arrays and Return Address Arrays

When the SYS$EXPREG system service adds pages to a region, it adds them in the normal direction of growth for the region. The return address array, if requested, indicates the order in which the pages were added. For example:

- If the program region is expanded, the starting virtual address is smaller than the ending virtual address.
- If the control region is expanded, the starting virtual address is larger than the ending virtual address.

The addresses returned indicate the first byte in the first page that was added or deleted and the last byte in the last page that was added or deleted, respectively.

When input address arrays are specified for the Create and Delete Virtual Address Space (SYS$CRETVA and SYS$DELTVA, respectively) system services, these services add or delete pages beginning with the address specified in the first longword and ending with the address specified in the second longword.

Alpha

On Alpha systems, the order in which the pages are added or deleted does not have to be in the normal direction of growth for the region. Moreover, because these services add or delete only whole pages, they ignore the low-order bits of the specified virtual address (the low-order bits contain the byte offset within the page). Table 20–1 shows the page size and byte offset.

Table 20–1 Page and Byte Offset Within Pages on Alpha Systems

Page Size (Bytes)	Byte Within Page (Bits)
8K	13
16K	14
32K	15
64K	16

Table 20–2 shows some sample virtual addresses in hexadecimal that may be specified as input to SYS$CRETVA or SYS$DELTVA and shows the return address arrays if all pages are successfully added or deleted. Table 20–2 assumes a page size of 8 KB = 2000 hex.

Table 20–2 Sample Virtual Address Arrays on Alpha Systems

Input Array		Output Array			
Start	End	Region	Start	End	Number of Pages
1010	1670	P0	0	1FFF	1
2450	2451	P0	2000	3FFF	1
4200	A500	P0	4000	BFFF	5
9450	9450	P0	8000	9FFF	1
7FFEC010	7FFEC010	P1	7FFEDFFF	7FFEC000	1
7FFEC010	7FFEBCA0	P1	7FFEDFFF	7FFEA000	2

For SYS$CRETVA and SYS$DELTVA, note that if the input virtual addresses are the same, as in the fourth and fifth items in Table 19–1, a single page is added or deleted. The return address array indicates that the page was added or deleted in the normal direction of growth for the region.

Note that for SYS$CRMPSC and SYS$MGBLSC, which are discussed in Section 20.4.7, the sample virtual address arrays in Table 19–1 do not apply. The reason is that the lower address value has to be an even multiple of the machine page size; that is, it must be rounded down to an even multiple page size. In addition, the higher address value must be one less than the even multiple page size, representing the last byte on the last page. That is, it must be rounded up to an even multiple page size, minus 1.

The procedure for determining start and end virtual addresses is as follows:

1. Get the page size in bytes.
2. Subtract 1 to get the byte-with-page mask.
3. Mask the low bits of lower virtual address, which is a round-down operation to round it to the next lower page boundary.
4. Perform a logical OR operation on the higher virtual address, which is a round-up operation to round it to the highest address in the last page. ♦

20.4.3 Allocating Memory in Existing Virtual Address Space on Alpha Systems (Alpha Only)

Alpha

On Alpha systems, if you reallocate memory that is already in its virtual address space by using the SYS$CRETVA system service, you may need to modify the values of the following arguments to SYS$CRETVA:

- If your application explicitly rounds the lower address specified in the **inadr** argument to be a multiple of 512 in order to align on a page boundary, you need to modify the address. The Alpha system's version of the SYS$CRETVA system service rounds down the start address to a CPU-specific page boundary, which will vary with different implementations. It also rounds up the end address to the last byte in a CPU-specific page boundary.
- The size of the reallocation, specified by the address range in the **inadr** argument, may be larger on an Alpha system than on a VAX system because the request is rounded up to CPU-specific pages. This can cause the unintended destruction of neighboring data, which may also occur with

single-page allocations. (When the start address and the end address specified in the **inadr** argument match, a single page is allocated.)

To determine whether you must modify the address as specified in **inadr**, specify the optional **retadr** argument to determine the exact boundaries of the memory allocated by the call to SYS$CRETVA. ♦

20.4.4 Page Ownership and Protection

Each page in the virtual address space of a process is owned by the access mode that created the page. For example, pages in the program region initially provided for the execution of an image are owned by user mode. Pages that the image creates dynamically are also owned by user mode. Pages in the control region, except for the pages containing the user stack, are normally owned by more privileged access modes.

Only the owner access mode or a more privileged access mode can delete the page or otherwise affect it. The owner of a page can also indicate, by means of a protection code, the type of access that each access mode will be allowed.

The Set Protection on Pages (SYS$SETPRT) system service changes the protection assigned to a page or group of pages. The protection is expressed as a code that indicates the specific type of access (none, read-only, read/write) for each of the four access modes (kernel, executive, supervisor, user). Only the owner access mode or a more privileged access mode can change the protection for a page.

When an image attempts to access a page that is protected against the access attempted, a hardware exception called an **access violation** occurs. When an image calls a memory management system service, the service probes the pages to be used to determine whether an access violation would occur if the image attempts to read or write one of the pages. If an access violation occurs, the service exits with the status code SS$_ACCVIO.

Because the memory management services add, delete, or modify a single page at a time, one or more pages can be successfully changed before an access violation is detected. If the **retadr** argument is specified in the service call, the service returns the addresses of pages changed (added, deleted, or modified) before the error. If no pages are affected, that is, if an access violation occurs on the first page specified, the service returns a −1 in both longwords of the return address array.

If the **retadr** argument is not specified, no information is returned.

20.4.5 Working Set Paging

Alpha

On Alpha systems, when a process is executing an image, a subset of its pages resides in physical memory; these pages are called the **working set** of the process. The working set includes pages in both the program region and the control region. The initial size of a process's working set is defined by the process's working set default (WSDEFAULT) quota, which is specified in pagelets. When ample physical memory is available, a process's working-set upper growth limit can be expanded to its working set extent (WSEXTENT). ♦

When the image refers to a page that is not in memory, a page fault occurs, and the page is brought into memory, possibly replacing an existing page in the working set. If the page that is going to be replaced is modified during the execution of the image, that page is written into a paging file on disk. When this page is needed again, it is brought back into memory, again replacing a current page from the working set. This exchange of pages between physical memory and secondary storage is called **paging**.

The paging of a process's working set is transparent to the process. However, if a program is very large or if pages in the program image that are used often are being paged in and out frequently, the overhead required for paging may decrease the program's efficiency. The SYS$ADJWSL, SYS$PURGWS, and SYS$LKWSET system services allow a process, within limits, to counteract these potential problems.

SYS$ADJWSL System Service

The Adjust Working Set Limit (SYS$ADJWSL) system service increases or decreases the maximum number of pages that a process can have in its working set. The format for this routine is as follows:

SYS$ADJWSL ([pagcnt],[wsetlm])

Alpha

On Alpha systems, use the **pagcnt** argument to specify the number of pagelets to add or subtract from the current working set size. The Alpha system rounds the specified number of pagelets to a multiple of the system's page size. The new working set size is returned in **wsetlm** in units of pagelets. ♦

SYS$PURGWS System Service

The Purge Working Set (SYS$PURGWS) system service removes one or more pages from the working set.

SYS$LKWSET System Service

The Lock Pages in Working Set (SYS$LKWSET) system service makes one or more pages in the working set ineligible for paging by locking them in the working set. Once locked into the working set, those pages remain in the working set until they are unlocked explicitly with the Unlock Pages in Working Set (SYS$ULWSET) system service, or program execution ends. The format is as follows:

SYS$LKWSET (inadr,[retadr],[acmode])

Alpha

Specifying a Range of Addresses On Alpha systems, use the **inadr** argument to specify the range of addresses to be locked. SYS$LKWSET rounds the addresses to CPU-specific page boundaries, if necessary. The range of addresses of the pages actually locked are returned in the **retadr** argument.

However, because the Alpha system's instructions cannot contain full virtual addresses, the Alpha system's images must reference procedures and data indirectly through a pointer to a procedure descriptor. The procedure descriptor contains information about the procedure, including the actual code address. These pointers to procedure descriptors and data are collected into a program section called a **linkage section**. Therefore, it is not sufficient simply to lock a section of code into memory to improve performance. You must also lock the associated linkage section into the working set.

To lock the linkage section into memory, you must determine the start and end addresses that encompass the linkage section and pass these addresses as values in the **inadr** argument to a call to SYS$LKWSET. For more information about linking, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*. ♦

Specifying the Access Mode Use the **acmode** argument to specify the access mode to be associated with the pages you want locked.

20.4.6 Process Swapping

The operating system balances the needs of all the processes currently executing, providing each with the system resources it requires on an as-needed basis. The memory management routines balance the memory requirements of the process. Thus, the sum of the working sets for all processes currently in physical memory is called the **balance set.**

When a process whose working set is in memory becomes inactive—for example, to wait for an I/O request or to hibernate—the entire working set or part of it may be removed from memory to provide space for another process's working set to be brought in for execution. This removal from memory is called **swapping.**

The working set may be removed in two ways:

- Partially—Also called **swapper trimming.** Pages are removed from the working set of the target process so that the number of pages in the working set is fewer, but the working set is not swapped.
- Entirely—Called swapping. All pages are swapped out of memory.

When a process is swapped out of the balance set, all the pages (both modified and unmodified) of its working set are swapped, including any pages that had been locked in the working set.

A privileged process may lock itself in the balance set. While pages can still be paged in and out of the working set, the process remains in memory even when it is inactive. To lock itself in the balance set, the process issues the Set Process Swap Mode (SYS$SETSWM) system service, as follows:

```
$SETSWM_S SWPFLG=#1
```

This call to SYS$SETSWM disables process swap mode. You can also disable swap mode by setting the appropriate bit in the STSFLG argument to the Create Process (SYS$CREPRC) system service; however, you need the PSWAPM privilege to alter process swap mode.

A process can also lock particular pages in memory with the Lock Pages in Memory (SYS$LCKPAG) system service. These pages are forced into the process's working set if they are not already there. When pages are locked in memory with this service, the pages remain in memory even when the remainder of the process's working set is swapped out of the balance set. These remaining pages stay in memory until they are unlocked with SYS$ULKPAG. The SYS$LCKPAG system service can be useful in special circumstances, for example, for routines that perform I/O operations to devices without using the operating system's I/O system.

You need the PSWAPM privilege to issue the SYS$LCKPAG or SYS$ULKPAG system service.

20.4.7 Sections

A **section** is a disk file or a portion of a disk file containing data or instructions that can be brought into memory and made available to a process for manipulation and execution. A section can also be one or more consecutive page frames in physical memory or I/O space; such sections, which require you to specify page frame number (PFN) mapping, are discussed in the Chapter 19, Section 19.5.6.15.

Sections are either private or global (shared).

- **Private sections** are accessible only by the process that creates them. A process can define a disk data file as a section, map it into its virtual address space, and manipulate it.
- **Global sections** can be shared by more than one process. One copy of the global section resides in physical memory, and each process sharing it refers to the same copy, except for copy-on-reference sections. For a copy-on-reference section, each process refers to the same global section, but each process gets its own copy of each page upon reference. A global section can contain shareable code or data that can be read, or read and written, by more than one process. Global sections are either temporary or permanent and can be defined for use within a group or on a systemwide basis. Global sections can be mapped to a disk file or created as a global page-file section, or they can be a PFN mapped section.

When modified pages in writable disk file sections are paged out of memory during image execution, they are written back into the section file rather than into the paging file, as is the normal case with files. (However, copy-on-reference sections are not written back into the section file.)

The use of disk file sections involves these two distinct operations:

1. The creation of a section defines a disk file as a section and informs the system what portions of the file contain the section.
2. The mapping of a section makes it available to a process and establishes the correspondence between virtual blocks in the file and specific addresses in the virtual address space of a process.

The Create and Map Section (SYS$CRMPSC) system service creates and maps a private section or a global section. Because a private section is used only by a single process, creation and mapping are simultaneous operations. In the case of a global section, one process can create a permanent global section and not map to it; other processes can map to it. A process can also create and map a global section in one operation.

The following sections describe the creation, mapping, and use of disk file sections. In each case, operations and requirements that are common to both private sections and global sections are described first, followed by additional notes and requirements for the use of global sections. Section 20.4.7.9 discusses global page-file sections.

20.4.7.1 Creating Sections

To create a disk file section, you must follow these steps:

1. Open or create the disk file containing the section.
2. Define which virtual blocks in the file comprise the section.
3. Define the characteristics of the section.

20.4.7.2 Opening the Disk File

Before you can use a file as a section, you must open it using OpenVMS RMS. The following example shows the OpenVMS RMS file access block ($FAB) and $OPEN macros used to open the file and the channel specification to the SYS$CRMPSC system service necessary for reading an existing file:

```
SECFAB: $FAB    FNM=<SECTION.TST>, ; File access block
                FOP=UFO
                RTV= -1
          .
          .
          .
        $OPEN   FAB=SECFAB
        $CRMPSC_S -
                CHAN=SECFAB+FAB$L_STV,...
```

The file options parameter (FOP) indicates that the file is to be opened for user I/O; this option is required so that OpenVMS RMS assigns the channel using the access mode of the caller. OpenVMS RMS returns the channel number on which the file is accessed; this channel number is specified as input to the SYS$CRMPSC system service (**chan** argument). The same channel number can be used for multiple create and map section operations.

The option RTV=−1 tells the file system to keep all of the pointers to be mapped in memory at all times. If this option is omitted, the SYS$CRMPSC service requests the file system to expand the pointer areas if necessary. Storage for these pointers is charged to the BYTLM quota, which means that opening a badly fragmented file can fail with an EXBYTLM failure status. Too many fragmented sections may cause the byte limit to be exceeded.

The file may be a new file that is to be created while it is in use as a section. In this case, use the $CREATE macro to open the file. If you are creating a new file, the file access block (FAB) for the file must specify an allocation quantity (ALQ parameter).

You can also use SYS$CREATE to open an existing file; if the file does not exist, it is created. The following example shows the required fields in the FAB for the conditional creation of a file:

```
GBLFAB: $FAB    FNM=<GLOBAL.TST>, -
                ALQ=4, -
                FAC=PUT,-
                FOP=<UFO,CIF,CBT>, -
                SHR=<PUT,UPI>
          .
          .
          .
        $CREATE FAB=GBLFAB
```

When the $CREATE macro is invoked, it creates the file GLOBAL.TST if the file does not currently exist. The CBT (contiguous best try) option requests that, if possible, the file be contiguous. Although section files are not required to be contiguous, better performance can result if they are.

20.4.7.3 Defining the Section Extents

After the file is opened successfully, the SYS$CRMPSC system service can create a section from the entire file or from only certain portions of it. The following arguments to SYS$CRMPSC define the extents of the file that comprise the section:

Alpha

- **pagcnt** (page count). On Alpha systems, this argument is optional. It indicates the size of the space that will be mapped. The **pagcnt** argument is in units of page frames (PFNs) for a PFN-mapped section and in units of pagelets (512-byte blocks) for disk-backed sections, including page file sections.

  If **pagcnt** is not supplied, then the section size defaults to the file size for a section being created and not mapped or defaults to the minimum of the file size and the size specified by **inadr** for a section being created and mapped simultaneously. You can map only what you have access to. Once a starting point is established, you can map **pagcnt** more space as long as you do not exceed the total size of the item you are mapping, such as the remaining blocks of a file or the remaining space in a global section. ♦
- **vbn** (virtual block number). This argument is optional. It defines the number of the virtual block in the file that is the beginning of the section. If you do not specify this argument, the value 1 is passed (the first virtual block in the file is the beginning of the section). If you have specified physical page frame number (PFN) mapping, the **vbn** argument specifies the starting PFN.

20.4.7.4 Defining the Section Characteristics

The **flags** argument to the SYS$CRMPSC system service defines the following section characteristics:

- Whether it is a private section or a global section. The default is to create a private section.
- How the pages of the section are to be treated when they are copied into physical memory or when a process refers to them. The pages in a section can be either or both of the following:
  - Read/write or read-only
  - Created as demand-zero pages or as copy-on-reference pages, depending on how the processes are going to use the section and whether the file contains any data (see Section 20.4.7.12)
- Whether the section is to be mapped to a disk file or to specific physical page frames (see Section 20.4.7.17).

Alpha

Table 20–3 shows the flag bits that must be set for specific characteristics on Alpha systems.

Table 20–3 Flag Bits to Set for Specific Section Characteristics on Alpha Systems

	Section to Be Created			
Correct Flag Combinations	Private	Global	PFN Private	PFN Global
SEC$M_GBL	0	1	0	1
SEC$M_CRF	Optional	Optional	0	0
SEC$M_DZRO	Optional	Optional	0	0
SEC$M_WRT	Optional	Optional	Optional	Optional
SEC$M_PERM	Not used	Optional	Not used	1
SEC$M_SYSGBL	Not used	Optional	Not used	Optional
SEC$M_PFNMAP	0	0	1	1
SEC$M_EXPREG	Optional	Optional	Optional	Optional
SEC$M_PAGFIL	0	Optional	0	0

When you specify section characteristics, the following restrictions apply:

- Global sections cannot be both demand-zero and copy-on-reference.
- Demand-zero sections must be writable. ♦

20.4.7.5 Defining Global Section Characteristics

If the section is a global section, you must assign a character string name (**gsdnam** argument) to it so that other processes can identify it when they map it. The format of this character string name is explained in Section 20.4.7.6.

The **flags** argument specifies the following types of global section:

- Group temporary (the default)
- Group permanent
- System temporary
- System permanent

Group global sections can be shared only by processes executing with the same group number. The name of a group global section is implicitly qualified by the group number of the process that created it. When other processes map it, their group numbers must match.

A temporary global section is automatically deleted when no processes are mapped to it, but a permanent global section remains in existence even when no processes are mapped to it. A permanent global section must be explicitly marked for deletion with the Delete Global Section (SYS$DGBLSC) system service.

You need the user privileges PRMGBL and SYSGBL to create permanent group global sections or system global sections (temporary or permanent), respectively.

A system global section is available to all processes in the system.

Optionally, a process creating a global section can specify a protection mask (**prot** argument) to restrict all access or a type of access (read, write, execute, delete) to other processes.

20.4.7.6 Global Section Name

The **gsdnam** argument specifies a descriptor that points to a character string.

Translation of the **gsdnam** argument proceeds in the following manner:

1. The current name string is prefixed with GBL$ and the result is subject to logical name translation.
2. If the result is a logical name, step 1 is repeated until translation does not succeed or until the number of translations performed exceeds the number specified by the system parameter LNM$C_MAXDEPTH.
3. The GBL$ prefix is stripped from the current name string that could not be translated. This current string is the global section name.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE GBL$GSDATA GSDATA_001
```

Your program contains the following statements:

```
#include <descrip.h>
   .
   .
   .
    $DESCRIPTOR(gsdnam,"GSDATA");
   .
   .
   .
    status = sys$crmpsc(&gsdnam,  . . .  );
```

The following logical name translation takes place:

1. GBL$ is prefixed to GSDATA.
2. GBL$GSDATA is translated to GSDATA_001. (Further translation is not successful. When logical name translation fails, the string is passed to the service.)

There are three exceptions to the logical name translation method discussed in this section:

- If the name string starts with an underscore (_), the operating system strips the underscore and considers the resultant string to be the actual name (that is, further translation is not performed).
- If the name string is the result of a logical name translation, then the name string is checked to see if it has the **terminal** attribute. If the name string is marked with the **terminal** attribute, the operating system considers the resultant string to be the actual name (that is, further translation is not performed).
- If the global section has a name in the format *name_nnn*, the operating system first strips the underscore and the digits *(nnn)*, then translates the resultant name according to the sequence discussed in this section, and finally reappends the underscore and digits. The system uses this method in conjunction with known images and shared files installed by the system manager.

20.4.7.7 Mapping Sections

When you call the SYS$CRMPSC system service to create or map a section, or both, you must provide the service with a range of virtual addresses (**inadr** argument) into which the section is to be mapped.

Alpha

On Alpha systems, the **inadr** argument specifies the size and location of the section by its start and end addresses. SYS$CRMPSC interprets the **inadr** argument in the following ways:

- If both addresses specified in the **inadr** argument are the same and the SEC$M_EXPREG bit is set in the **flags** argument, SYS$CRMPSC allocates the memory in whichever program region the addresses fall but does not use the specified location.
- If both addresses are different, SYS$CRMPSC maps the section into memory using the boundaries specified.

If you know specifically which pages the section should be mapped into, you provide these addresses in a 2-longword array. For example, to map a private section of 10 pages into virtual pages 10 through 19 of the program region, specify the input address array as follows: ♦

```
unsigned int maprange[1]; /* Assume page size = 8 KB */

maprange[0] = 0x14000;    /* Address (hex) of page 10 */
maprange[1] = 0x27FFF;    /* Address (hex) of page 19 */
```

Alpha

On Alpha systems, the **inadr** argument range must have a lower address on an even page boundary and a higher address exactly one less than a page boundary. For example, the range can be expressed as the following on an 8 KB page system:

0 —-> 1FFF
2000 —-> 7FFF
or
inadr[0] = first byte in range
inadr[1] = last byte in range

If the range is not expressed in terms of page-inclusive boundaries, then an SS$_ INVARG condition value is returned. ♦

You do not need to know the explicit addresses to provide an input address range. If you want the section mapped into the first available virtual address range in the program region (P0) or control region (P1), you can specify the SEC$M_ EXPREG flag bit in the **flags** argument. In this case, the addresses specified by the **inadr** argument control whether the service finds the first available space in the P0 or P1. The value specified or defaulted for the **pagcnt** argument determines the amount of space mapped.

Alpha

On Alpha systems, the **relpag** argument specifies the location in the section file at which you want mapping to begin.

The SYS$CRMPSC and SYS$MGBLSC system services map a minimum of one CPU-specific page. If the section file does not fill a single page, the remainder of the page is filled with zeros after faulting the page into memory. The extra space on the page should not be used by your application because only the data that fits into the section file will be written back to the disk.

The following example shows part of a program used to map a section at the current end of the program region:

```
    unsigned int status, inadr[1], retadr[1], flags;

/*
    This range used merely to indicate P0 space since SEC$M_EXPREG
       is specified
*/
    inadr[0]= 0x200; /* Any program (P0) region address */
    inadr[1]= 0x200; /* Any P0 address (can be same) */

   .
   .
   .
/* Address range returned in retadr */

    flags = SEC$M_EXPREG;
    status = sys$crmpsc(&inadr, &retadr, flags,  . . .  );
```

The addresses specified do not have to be currently in the virtual address space of the process. The SYS$CRMPSC system service creates the required virtual address space during the mapping of the section. If you specify the **retadr** argument, the service returns the range of addresses actually mapped. ♦

Alpha

On Alpha systems, the starting **retadr** address should match **inadr**, plus **relpag** if specified. The ending (higher) address will be limited by the lower of:

- The value of the **pagcnt** argument
- The actual remaining block count in the file starting with specified starting **vbn**, or **relpag**
- The bound dictated by the **inadr** argument ♦

After a section is mapped successfully, the image can refer to the pages using one of the following:

- A base register or pointer and predefined symbolic offset names
- Labels defining offsets of an absolute program section or structure

Alpha

The following example shows part of a program used to create and map a process section on Alpha systems:

```
SECFAB: $FAB    FNM=<SECTION.TST>, -
                FOP=UFO, -
                FAC=PUT, -
                SHR=<GET,PUT,UPI>
;
MAPRANGE:
        .LONG   ^X14000                  ; First 8 KB page
        .LONG   ^X27FFF                  ; Last page
RETRANGE:
        .BLKL   1                       ; First page mapped
ENDRANGE:
        .BLKL   1                       ; Last page mapped
          .
          .
          .
        $OPEN   FAB=SECFAB              ; Open section file
        BLBS    R0,10$
        BSBW    ERROR
```

```
10$:    $CRMPSC_S -
                INADR=MAPRANGE,-        ; Input address array
                RETADR=RETRANGE,-       ; Output array
                PAGCNT=#4,-             ; Map four pagelets
                FLAGS=#SEC$M_WRT,-      ; Read/write section
                CHAN=SECFAB+FAB$L_STV   ; Channel number
        BLBS    R0,20$
        BSBW    ERROR
20$:    MOVL    RETRANGE,R6             ; Point to start of section
```

Notes on Example

1. The OPEN macro opens the section file defined in the file access block SECFAB. (The FOP parameter to the $FAB macro must specify the UFO option.)
2. The SYS$CRMPSC system service uses the addresses specified at MAPRANGE to specify an input range of addresses into which the section will be mapped. The **pagcnt** argument requests that only 4 pagelets of the file be mapped.
3. The **flags** argument requests that the pages in the section have read/write access. The symbolic flag definitions for this argument are defined in the $SECDEF macro. Note that the file access field (FAC parameter) in the FAB also indicates that the file is to be opened for writing.
4. When SYS$CRMPSC completes, the addresses of the 4 pagelets that were mapped are returned in the output address array at RETRANGE. The address of the beginning of the section is placed in register 6, which serves as a pointer to the section. ♦

20.4.7.8 Mapping Global Sections

A process that creates a global section can map that global section. Then other processes can map it by calling the Map Global Section (SYS$MGBLSC) system service.

When a process maps a global section, it must specify the global section name assigned to the section when it was created, whether it is a group or system global section, and whether it desires read-only or read/write access. The process may also specify the following:

- A version identification (**ident** argument), indicating the version number of the global section (when multiple versions exist) and whether more recent versions are acceptable to the process.
- A relative pagelet number (**relpag** argument), specifying the pagelet number, relative to the beginning of the section, to begin mapping the section. In this way, processes can use only portions of a section. Additionally, a process can map a piece of a section into a particular address range and subsequently map a different piece of the section into the same virtual address range.

Alpha

On Alpha systems, you should specify the **retadr** argument to determine the exact boundaries of the memory that was mapped by the call. If your application specifies the **relpag** argument, you *must* specify the **retadr** argument. In this case, it is not an optional argument. ♦

Cooperating processes can both issue a SYS$CRMPSC system service to create and map the same global section. The first process to call the service actually creates the global section; subsequent attempts to create and map the section result only in mapping the section for the caller. The successful return status code SS$_CREATED indicates that the section did not already exist when the

SYS$CRMPSC system service was called. If the section did exist, the status code SS$_NORMAL is returned.

The example in Section 20.4.7.12 shows one process (ORION) creating a global section and a second process (CYGNUS) mapping the section.

20.4.7.9 Global Page-File Sections

Global page-file sections are used to store temporary data in a global section. A global page-file section is a section of virtual memory that is not mapped to a file. The section can be deleted when processes have finished with it. (Contrast this with demand-zero section file pages where no initialization is necessary, but the pages are saved in a file.) The system parameter GBLPAGFIL controls the total number of global page-file pages in the system.

To create a global page-file section, you must set the flag bits SEC$M_GBL and SEC$M_PAGFIL in the **flags** argument to the Create and Map Section (SYS$CRMPSC) system service. The channel (**chan** argument) must be 0.

You cannot specify the flag bit SEC$M_CRF with the flag bit SEC$M_PAGFIL.

20.4.7.10 Mapping into a Defined Address Range

Alpha

On Alpha systems, SYS$CRMPSC and SYS$MGBLSC interpret some of the arguments differently than on VAX systems if you are mapping a section into a defined area of virtual address space. The differences are as follows:

- The addresses specified as values in the **inadr** argument must be aligned on CPU-specific page boundaries. On VAX systems, SYS$CRMPSC and the SYS$MGBLSC round these addresses to page boundaries for you. On Alpha systems, SYS$CRMPSC does not round the addresses you specify to page boundaries, because rounding to CPU-specific page boundaries on Alpha system affects a much larger portion of memory than it does on VAX systems, where page sizes are much smaller. Therefore, on Alpha systems, you must explicitly state where you want the virtual memory space mapped. If the addresses you specify are not aligned on CPU-specific page boundaries, SYS$CRMPSC returns an invalid arguments error (SS$_INVARG).

  In particular, the lower **inadr** address must be on a CPU-specific page boundary, and the higher **inadr** address must be one less than a CPU-specific page; that is, it indicates the highest-addressed byte of the **inadr** range.

- The addresses returned in the **retadr** argument reflect only the usable portion of the actual memory mapped by the call, not the entire amount mapped. The usable amount is either the value specified in the **pagcnt** argument (measured in pagelets) or the size of the section file, whichever is smaller. The actual amount mapped depends on how many CPU-specific pages are required to map the section file. If the section file does not fill a CPU-specific page, the remainder of the page is filled with zeros. The excess space on this page should not be used by your application. The end address specified in the **retadr** argument specifies the upper limit available to your application. Also note that, when the **relpag** argument is specified, the **retadr** argument *must* be included. It is not optional on Alpha systems. ♦

20.4.7.11 Mapping from an Offset into a Section File

Alpha

On Alpha systems, you can map a portion of a section file by specifying the address at which to start the mapping as an offset from the beginning of the section file. You specify this offset by supplying a value to the **relpag** argument of SYS$CRMPSC. The value of the **relpag** argument specifies the pagelet number relative to the beginning of the file at which the mapping should begin.

To preserve compatibility, SYS$CRMPSC interprets the value of the **relpag** argument in 512-byte units on both VAX systems and Alpha systems. However, because the CPU-specific page size on the Alpha system is larger than 512 bytes, the address specified by the offset in the **relpag** argument probably does not fall on a CPU-specific page boundary on an Alpha system. SYS$CRMPSC can map virtual memory in CPU-specific page increments only. Therefore, on Alpha systems, the mapping of the section file will start at the beginning of the CPU-specific page that contains the offset address, not at the address specified by the offset.

Note

Even though the routine starts mapping at the beginning of the CPU-specific page that contains the address specified by the offset, the start address returned in the **retadr** argument is the address specified by the offset, not the address at which mapping actually starts.

If you map from an offset into a section file, you must still provide an **inadr** argument that abides by the requirements presented in Section 20.4.7.10 when mapping into a defined address range. ♦

20.4.7.12 Section Paging

The first time an image executing in a process refers to a page that was created during the mapping of a disk file section, the page is copied into physical memory. The address of the page in the virtual address space of a process is mapped to the physical page. During the execution of the image, normal paging can occur; however, pages in sections are not written into the page file when they are paged out, as is the normal case. Rather, if they have been modified, they are written back into the section file on disk. The next time a page fault occurs for the page, the page is brought back from the section file.

If the pages in a section were defined as demand-zero pages or copy-on-reference pages when the section was created, the pages are treated differently, as follows:

- If the call to SYS$CRMPSC requested that pages in the section be treated as demand-zero pages, these pages are initialized to zeros when they are created in physical memory. If the file is either a new file being created as a section or a file being completely rewritten, demand-zero pages provide a convenient way of initializing the pages. The pages are paged back into the section file.
- When the virtual address space is deleted, all unreferenced pages are written back to the file as zeros. This causes the file to be initialized, no matter how few pages were modified.
- If the call to SYS$CRMPSC requested that pages in the section be copy-on-reference pages, each process that maps to the section receives its own copy of the section, on a page-by-page basis from the file, as it refers to them. These pages are never written back into the section file but are paged to the paging file as needed.

In the case of global sections, more than one process can be mapped to the same physical pages. If these pages need to be paged out or written back to the disk file defined as the section, these operations are done only when the pages are not in the working set of any process.

Alpha

In the following example for Alpha systems, process ORION creates a global section and process CYGNUS maps to that section:

```
/* Process ORION */

#include <rms.h>
#include <rmsdef.h>
#include <string.h>
#include <secdef.h>
#include <descrip.h>

struct FAB gblfab;

main() {
    unsigned short chan;
    unsigned int status, flags, efn=65;
    char *fn = "SECTION.TST";
    $DESCRIPTOR(name, "FLAG_CLUSTER");     /* Common event flag cluster name */
    $DESCRIPTOR(gsdnam, "GLOBAL_SECTION"); /* Global section name */

1 status = SYS$ASCEFC(efn, &name, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

/* Initialize FAB fields */

    gblfab = cc$rms_fab;
    gblfab.fab$l_alq = 4;
    gblfab.fab$b_fac = FAB$M_PUT;
    gblfab.fab$l_fnm = fn;
    gblfab.fab$l_fop = FAB$M_CIF || FAB$M_CBT;

   .
   .
   .

/* Create a file if none exists */

2 status = SYS$CREATE( &gblfab, 0, 0 );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    flags = SEC$M_GBL | SEC$M_WRT;
    status = SYS$CRMPSC(0, 0, 0, flags, &gsdnam,  . . .  );
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$SETEF(efn);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );
   .
   .
   .
}

/* Process CYGNUS */

    unsigned int status, efn=65;
    $DESCRIPTOR(cluster,"FLAG_CLUSTER");
    $DESCRIPTOR(section,"GLOBAL_SECTION");
   .
   .
   .
```

```
3 status = SYS$ASCEFC(efn, &cluster, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$WAITFR(efn);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

    status = SYS$MGBLSC(&inadr, &retadr, 0, flags, &section, 0, 0);
    if ((status & 1) != 1)
        LIB$SIGNAL( status );

}
```

1 The processes ORION and CYGNUS are in the same group. Each process first associates with a common event flag cluster named FLAG_CLUSTER to use common event flags to synchronize its use of the section.

2 The process ORION creates the global section named GLOBAL_SECTION, specifying section flags that indicate that it is a global section (SEC$M_GBL) and has read/write access. Input and output address arrays, the page count parameter, and the channel number arguments are not shown; procedures for specifying them are the same, as shown in this example.

3 The process CYGNUS associates with the common event flag cluster and waits for the flag defined as FLGSET; ORION sets this flag when it has finished creating the section. To map the section, CYGNUS specifies the input and output address arrays, the flag indicating that it is a global section, and the global section name. The number of pages mapped is the same as that specified by the creator of the section. ♦

20.4.7.13 Reading and Writing Data Sections

Read/write sections provide a way for a process or cooperating processes to share data files in virtual memory.

The sharing of global sections may involve application-dependent synchronization techniques. For example, one process can create and map to a global section in read/write fashion; other processes can map to it in read-only fashion and interpret data written by the first process. Alternatively, two or more processes can write to the section concurrently. (In this case, the application must provide the necessary synchronization and protection.)

After a file is updated, the process or processes can release (or unmap) the section. The modified pages are then written back into the disk file defined as a section.

When this is done, the revision number of the file is incremented, and the version number of the file remains unchanged. A full directory listing indicates the revision number of the file and the date and time that the file was last updated.

20.4.7.14 Releasing and Deleting Sections

A process unmaps a section by deleting the virtual addresses in its own virtual address space to which it has mapped the section. If a return address range was specified to receive the virtual addresses of the mapped pages, this address range can be used as input to the Delete Virtual Address Space (SYS$DELTVA) system service, as follows:

```
$DELTVA_S INADR=RETRANGE
```

When a process unmaps a private section, the section is deleted; that is, all control information maintained by the system is deleted. A temporary global section is deleted when all processes that have mapped to it have unmapped it. Permanent global sections are not deleted until they are specifically marked for deletion with the Delete Global Section (SYS$DGBLSC) system service; they are then deleted when no more processes are mapped.

Note that deleting the pages occupied by a section does not delete the section file, but rather cancels the process's association with the file. Moreover, when a process deletes pages mapped to a read/write section and no other processes are mapped to it, all modified pages are written back into the section file.

After a section is deleted, the channel assigned to it can be deassigned. The process that created the section can deassign the channel with the Deassign I/O Channel (SYS$DASSGN) system service, as follows:

```
$DASSGN_S CHAN=GBLFAB+FAB$L_STV
```

20.4.7.15 Writing Back Sections

Because read/write sections are not normally updated on disk until the physical pages they occupy are paged out, or until all processes referring to the section have unmapped it, a process should ensure that all modified pages are successfully written back into the section file at regular intervals.

The Update Section File on Disk (SYS$UPDSEC) system service writes the modified pages in a section into the disk file. The SYS$UPDSEC system service is described in the *OpenVMS System Services Reference Manual*.

20.4.7.16 Image Sections

Global sections can contain shareable code. The operating system uses global sections to implement shareable code, as follows:

1. The object module containing code to be shared is linked to produce a shareable image. The shareable image is not, in itself, executable. It contains a series of sections, called **image sections.**
2. You link private object modules with the shareable image to produce an executable image. No code or data from the shareable image is put into the executable image.
3. The system manager uses the INSTALL command to create a permanent global section from the shareable image file, making the image sections available for sharing.
4. When you run the executable image, the operating system automatically maps the global sections created by the INSTALL command into the virtual address space of your process.

For details on how to create and identify shareable images and how to link them with private object modules, see the *OpenVMS Linker Utility Manual*. For information about how to install shareable images and make them available for sharing as global sections, see the *OpenVMS System Manager's Manual*.

20.4.7.17 Page Frame Sections

A **page frame section** is one or more contiguous pages of physical memory or I/O space that have been mapped as a section. One use of page frame sections is to map to an I/O page, thus allowing a process to read device registers.

A page frame section differs from a disk file section in that it is not associated with a particular disk file and is not paged. However, it is similar to a disk file section in most other respects: you create, map, and define the extent and characteristics of a page frame section in essentially the same manner as you do for a disk file section.

To create a page frame section, you must specify page frame number (PFN) mapping by setting the SEC$M_PFNMAP flag bit in the **flags** argument to the Create and Map Section (SYS$CRMPSC) system service. The **vbn** argument is now used to specify that the first page frame is to be mapped instead of the first virtual block. You must have the user privilege PFNMAP to create or delete a page frame section but not to map to an existing one.

Because a page frame section is not associated with a disk file, you do not use the **chan**, and **pfc** arguments to the SYS$CRMPSC service to create or map this type of section. For the same reason, the SEC$M_CRF (copy-on-reference) and SEC$M_DZRO (demand-zero) bit settings in the **flags** argument do not apply. Pages in page frame sections are not written back to any disk file (including the paging file). The **pagcnt** and **relpag** arguments are in units of CPU-specific pages for page frame sections.

Caution

You must use caution when working with page frame sections. If you permit write access to the section, each process that writes to it does so at its own risk. Serious errors can occur if a process writes incorrect data or writes to the wrong page, especially if the page is also mapped by the system or by another process. Thus, any user who has the PFNMAP privilege can damage or violate the security of a system.

20.4.7.18 Partial Sections

Alpha

On Alpha systems, a **partial section** is one where not all of the defined section, whether private or global, is entirely backed up by disk blocks. In other words, a partial section is where a disk file does not map completely onto an Alpha system page.

For example, suppose a file for which you wish to create a section consists of 17 virtual blocks on disk. To map this section, you would need two whole Alpha 8 KB pages, the smallest size Alpha page available. The first Alpha page would map the first 16 blocks of the section, and the second Alpha page would map the 17th block of the section. (A block on disk is 512 bytes, same as on OpenVMS VAX.) This results in 15/16ths of the second Alpha page not being backed up by the section file. This is called a partial section because the second Alpha page of the section is only partially backed up.

When the partial page is faulted in, a disk read is issued for only as many blocks as actually back up that page, which in this case is 1. When that page is written back, only the one block is actually written.

If the upper portion of the second Alpha page is used, it is done so at some risk, because only the first block of that page is saved on a write-back operation. This upper portion of the second Alpha page is not really useful space to the programmer, because it is discarded during page faulting. ♦

20.4.8 Example of Using Memory Management System Services

Alpha

In the following example, two programs are communicating through a global section. The first program creates and maps a global section (by using SYS$CRMPSC) and then writes a device name to the section. This program also defines the device terminal and process names and sets the event flags that synchronize the processes.

The second program maps the section (by using SYS$MGBLSC) and then reads the device name and the process that allocated the device and any terminal allocated to that process. This program also writes the process named to the terminal global section where the process name can be read by the first program.

The common event cluster is used to synchronize access to the global section. The first program sets REQ_FLAG to indicate that the device name is in the section. The second program sets INFO_FLAG to indicate that the process and terminal names are available.

Data in a section must be page aligned. The following is the option file used at link time that causes the data in the common area named DATA to be page aligned:

PSECT_ATTR = DATA, PAGE

For high-level language usage, use the **solitary** attribute of the linker. See the *OpenVMS Linker Utility Manual* for an explanation of how to use the **solitary** attribute. The address range requested for a section must end on a page boundary, so SYS$GETSYI is used to obtain the system page size.

Before executing the first program, you need to write a user-open routine that sets the user-open bit (FAB$V_UFO) of the FAB options longword (FAB$L_FOP). Because the Fortran OPEN statement specifies that the file is new, $CREATE should be used to open it rather than $OPEN. No $CONNECT should be issued. The user-open routine reads the channel number that the file is opened on from the status longword (FAB$L_STV) and returns that channel number to the main program by using a common block (CHANNEL in this example).

```
!This is the program that creates the global section.

! 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
! (returned from useropen routine)
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/

! Data for SYS$GETSYI
INTEGER   PAGE_SIZE
INTEGER*2 BUFF_LEN, ITEM_CODE
INTEGER   BUFF_ADDR, LENGTH, TERMINATOR
EXTERNAL  SYI$_PAGE_SIZE
COMMON /GETSYI_ITEMLST/ BUFF_LEN,
2                       ITEM_CODE,
2                       BUFF_ADDR,
2                       LENGTH,
2                       TERMINATOR

! 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 element - first element + size 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 the system page size
BUFF_LEN = 4
ITEM_CODE = %LOC(SYI$_PAGE_SIZE)
BUFF_ADDR = %LOC(PAGE_SIZE)
LENGTH = 0
TERMINATOR = 0

STATUS = SYS$GETSYI(,,,BUFF_LEN,,,)
```

```
! Get location of data
PASS_ADDR (1) = %LOC (DEVICE)
PASS_ADDR (2) = PASS_ADDR(1) + PAGE_SIZE - 1

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 = '$DISK1'

! 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))
.
.
.

! This is the program that maps to the global section
! created by the previous program.

! 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/
```

```
! Data for SYS$GETSYI
INTEGER   PAGE_SIZE
INTEGER*2 BUFF_LEN, ITEM_CODE
INTEGER   BUFF_ADDR, LENGTH, TERMINATOR
EXTERNAL  SYI$_PAGE_SIZE
COMMON /GETSYI_ITEMLST/ BUFF_LEN,
2                       ITEM_CODE,
2                       BUFF_ADDR,
2                       LENGTH,
2                       TERMINATOR
.
.
.
! Get the system page size
BUFF_LEN = 4
ITEM_CODE = %LOC(SYI$_PAGE_SIZE)
BUFF_ADDR = %LOC(PAGE_SIZE)
LENGTH = 0
TERMINATOR = 0

STATUS = SYS$GETSYI(,,,BUFF_LEN,,,)

! 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) = PASS_ADDR(1) + PAGE_SIZE - 1

! 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
```

♦

21 Using Run-Time Routines for Memory Allocation

This chapter describes the use of run-time routines (RTLs) to allocate and deallocate pages. It contains the following sections:

Section 21.1 describes the allocating and freeing of pages.

Section 21.2 describes the interactions with other RTL routines.

Section 21.3 describes the interactions with system services.

Section 21.4 describes how to use zones.

Section 21.5 describes the mechanism for allocating and freeing blocks of memory.

Section 21.6 describes the RTL algorithms used to allocate and free memory.

Section 21.7 describes how to create and manage user-defined zones.

Section 21.8 describes the methods of debugging programs that use virtual memory zones.

Note

In this chapter, all references to *pages* include both the 512-byte page size on VAX systems and the 512-byte pagelet size on Alpha systems. See Chapter 19 and Chapter 20 for a discussion of page sizes on VAX and Alpha systems.

21.1 Allocating and Freeing Pages

The run-time library page management routines LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE provide a flexible mechanism for allocating and freeing pages (pagelets on Alpha systems) of memory. In general, modular routines should use these routines rather than direct system service calls to manage memory. The page or pagelet management routines maintain a processwide pool of free pages or pagelets and automatically reuse free pages or pagelets. If your program calls system services directly, it must do the bookkeeping to keep track of free memory.

LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE are fully reentrant. They can be called by code running at AST level or in an Ada multitasking environment.

Memory space allocated by LIB$GET_VM_PAGE are created with user-mode read-write access, even if the call to LIB$GET_VM_PAGE is made from a more privileged access mode.

LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE are designed for request sizes ranging from one page to a few hundred pages. If you are using very large request sizes of contiguous space in a single request, the bitmap allocation method that is used may cause fragmentation of your virtual address space because allocated pages are contiguous. For very large request sizes, use direct calls to SYS$EXPREG and do not use LIB$GET_VM_PAGE.

The format for LIB$GET_VM_PAGE is as follows:

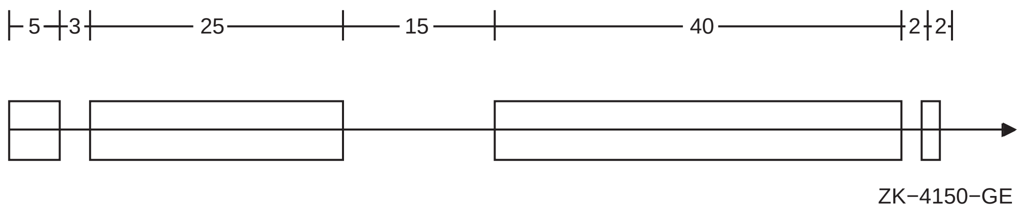
LIB$GET_VM_PAGE (num-pages, base-adr, [zone-id])

With this routine, you need to specify only the number of pages you need in the **num-pages** argument. The routine returns the base address of the contiguous block of pages that have been allocated in the **base-adr** argument.

The rules for using LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE are as follows:

- Any memory you free by calling LIB$FREE_VM_PAGE must have been allocated by a previous call to LIB$GET_VM_PAGE. You cannot allocate memory by calling SYS$EXPREG or SYS$CRETVA and then free it using LIB$FREE_VM_PAGE.
- All memory allocated by LIB$GET_VM_PAGE is page aligned; that is, the low-order 9 bits of the address are all zero. All memory freed by LIB$FREE_VM_PAGE must also be page aligned; an error status is returned if you attempt to free a block of memory that is not page aligned.
- You can free a smaller group of pages than you allocated. That is, if you allocated a group of 4 contiguous pages by a single call to LIB$GET_VM_PAGE, you can free the memory by using several calls to LIB$FREE_VM_PAGE; for example, free 1 page, 2 pages, and 1 page.
- You can combine contiguous groups of pages that were allocated by several calls to LIB$GET_VM_PAGE into one group of pages that are freed by a single call to LIB$FREE_VM_PAGE. Before doing this, however, you must compare the addresses to ensure that the pages you are combining are indeed contiguous. Of course, you must ensure that a routine frees only pages that it has previously allocated and still owns.
- Be especially careful that you do not attempt to free a set of pages twice. You might free a set of pages in one routine and reallocate those same pages from another routine. If the first routine then deallocates those pages a second time, any information that the second routine stored in them is lost. Because the pages are still allocated to your program (even though to a different routine), this type of programming mistake does not generate an error.
- The contents of memory allocated by LIB$GET_VM_PAGE are unpredictable. Your program must assign values to all locations that it uses.
- You should try to minimize the number of request sizes your program uses to avoid fragmentation of the free page pool. This concept is shown in Figure 21–1.

Figure 21–1 Memory Fragmentation



The straight line running across Figure 21–1 represents the memory allocated to your program. The blocks represent memory that has already been allocated. At this point, if you request 16 pages, memory will have to be allocated at the far right end of the memory line shown in this figure, even though there are 20 free pages before that point. You cannot use 16 of these 20 pages because the 20 free pages are “fragmented” into groups of 15, 3, and 2 pages.

Fragmentation is discussed further in Section 21.4.1.

21.2 Interactions with Other Run-Time Library Routines

Chapter 19 and Chapter 20 describe a three-level hierarchy of memory allocation routines consisting of the following:

1. Memory management system services
2. Run-time library page management routines LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE
3. Run-time library heap management routines LIB$GET_VM and LIB$FREE_VM

The run-time library and various programming languages provide another level of more specialized allocation routines.

- The run-time library dynamic string package provides a set of routines for allocating and freeing dynamic strings. The set of routines includes the following:

  LIB$SGET1_DD, LIB$SFREE1_DD
  LIB$SFREEN_DD
  STR$GET1_DX, STR$FREE1_DX

- VAX Ada provides allocators and the UNCHECKED_DEALLOCATION package for allocating and freeing memory.
- VAX Pascal provides the NEW and DISPOSE routines for allocating and freeing memory.
- VAX PL/I provides ALLOCATE and FREE statements for allocating and freeing memory.

A program containing routines written in several operating system languages may use a number of these facilities at the same time. This does not cause any problems or impose any restrictions on the user because all of these are layered on the run-time library heap management routines.

Note

To ensure correct operation, memory that is allocated by one of the higher-level allocators in the preceding list can be freed only by using the corresponding deallocation routine. That is, memory allocated by PASCAL NEW must be freed by calling PASCAL DISPOSE, and a dynamic string can be freed only by calling one of the string package deallocation routines.

21.3 Interactions with System Services

The run-time library page management and heap management routines are implemented as layers built on the memory management system services. In general, modular routines should use the run-time library routines rather than directly call memory management system services. However, in some situations you must use both. This section describes relationships between the run-time library and memory management. See the *OpenVMS System Services Reference Manual* for descriptions of the memory management system services.

You can use the Expand Region (SYS$EXPREG) system service to create pages of virtual memory in the program region (P0 space) for your process. The operating system keeps track of the first free page address at the end of P0 space, and it updates this free page address whenever you call SYS$EXPREG or SYS$CRETVA. The LIB$GET_VM_PAGE routine calls SYS$EXPREG to create pages, so there is no conflicting address assignments when you call SYS$EXPREG directly.

Avoid using the Create Virtual Address Space (SYS$CRETVA) system service, because you must specify the range of virtual addresses when it is called. If the address range you specify contains pages that already exist, SYS$CRETVA deletes those pages and recreates them as demand-zero pages. You may have difficulty avoiding conflicting address assignments if you use run-time library routines and SYS$CRETVA.

You must not use the Contract Region (SYS$CNTREG) system service, because other routines or the OpenVMS Record Management Services (RMS) may have allocated pages at the end of the program region.

You can change the protection on pages your program has allocated by calling the Set Protection (SYS$SETPRT) system service. All pages allocated by LIB$GET_VM_PAGE have user-mode read/write access. If you change protection on pages allocated by LIB$GET_VM_PAGE, you must reset the protection to user-mode read/write before calling LIB$FREE_VM_PAGE to free the pages.

You can use the Create and Map Section (SYS$CRMPSC) system service to map a file into your virtual address space. To map a file, you provide a range of virtual addresses for the file. One way to do this is to specify the Expand Region option (SEC$M_EXPREG) when you call SYS$CRMPSC. This method assigns addresses at the end of P0 space, similar to the SYS$EXPREG system service. Alternatively, you can provide a specific range of virtual addresses when you call SYS$CRMPSC; this is similar to allocating pages by calling SYS$CRETVA. If you assign a specific range of addresses, you must avoid conflicts with other routines. One way to do this is to allocate memory by calling LIB$GET_VM_PAGE and then use that memory to map the file.

The complete sequence of steps is as follows:

1. Call LIB$GET_VM_PAGE to allocate a contiguous group of (*n*+1) pages. The first *n* pages is used to map the file; the last page serves as a guard page.
2. Call SYS$CRMPSC using the first *n* pages to map the file into your process address space.
3. Process the file.
4. Call SYS$DELTVA to delete the first *n* pages and unmap the file.
5. Call SYS$CRETVA to recreate the *n* pages of virtual address space as demand-zero pages.
6. Call LIB$FREE_VM_PAGE to free (*n*+1) pages of memory and return them to the processwide page pool.

This sequence is satisfactory when mapping small files of a few hundred pages, but it has severe limitations when mapping very large files. As discussed in Section 21.1, you should not use LIB$GET_VM_PAGE to allocate very large groups of contiguous pages in a single request. In addition, when you allocate memory by calling LIB$GET_VM_PAGE (and thus SYS$EXPREG), the pages are charged against your process page file quota. Your page file quota is not charged if you call SYS$CRMPSC with the SEC$M_EXPREG option.

You can process very large files using SYS$CRMPSC by first providing a pool of pages that is sufficient for your program and then by using SYS$CRMPSC and SYS$DELTVA to map and unmap the file. Use LIB$SHOW_VM to obtain an estimate of how much dynamically allocated memory your program requires; round this number up and allow for increased memory usage in the future. You can then use the memory estimate as follows:

1. At the beginning of your program, include code to call LIB$GET_VM_PAGE and allocate the estimated number of pages. You should not request a large number of pages in one call to LIB$GET_VM_PAGE, because this would require contiguous allocation of the pages.
2. Call LIB$FREE_VM_PAGE to free all the pages allocated in step 1; this establishes a pool of free pages for your program.
3. Open files that your program needs; note that RMS may allocate buffers in P0 space.
4. Call SYS$CRMPSC specifying SEC$M_EXPREG to map the file into your process address space at the end of P0 space.
5. Process the file.
6. Call SYS$DELTVA, specifying the address range to release the file. If additional pages were not created after you mapped the file, SYS$DELTVA contracts your address space. Your program can repeat the process of mapping a file without continually expanding its address space.

21.4 Zones

The run-time library heap management routines LIB$GET_VM and LIB$FREE_VM are based on the concept of zones. A **zone** is a logically independent memory pool or subheap that you can control as one unit. A program may use several zones to structure its heap memory management. You might use a zone to:

- Store short-lived data structures that you can subsequently delete all at once
- Store a program that does not reference a wide range of addresses
- Specify a memory allocation algorithm specific to your program
- Specify attributes, like block size and alignment, specific to your program

You create a zone with specified attributes by calling the routine LIB$CREATE_VM_ZONE. LIB$CREATE_VM_ZONE returns a zone identifier value that you can use in subsequent calls to the routines LIB$GET_VM and LIB$FREE_VM. When you no longer need the zone, you can delete the zone and free all the memory it controls by a single call to LIB$DELETE_VM_ZONE.

The format for this routine is as follows:

```
LIB$CREATE_VM_ZONE (zone_id,[algorithm],[algorithm_arg] [,flags]
                        [,extend_size],[initial_size]
                        ,[block_size],[alignment],[page_limit],[p1])
```

For more information about LIB$CREATE_VM_ZONE, refer to the *OpenVMS RTL Library (LIB$) Manual*.

Allocating Address Space

Use the **algorithm** argument to specify how much space should be allocated—as a linked list of free blocks, as a set of lookaside list indexes by request size, as a set of lookaside lists for some block sizes, or as a single queue of free blocks.

Allocating Pages Within the Zone

Use the **initial_size** argument to allocate a specified number of pages from the zone when it is created. After zone creation, you can use LIB$GET_VM to allocate space.

Specifying the Block Size

Use the **block_size** argument to specify, in bytes, the block size.

Specifying Block Alignment

Use the **alignment** argument to specify, in bytes, the alignment for each block allocated.

Once a zone has been created and used, use LIB$DELETE_VM_ZONE to delete the zone and return the pages allocated to the processwide page pool. LIB$RESET_VM_ZONE frees pages for subsequent allocation but does not delete the zone or return the pages to the processwide page pool. Use LIB$SHOW_VM_ZONE to get information about a specific zone.

If you want a program to deal with each VM zone created during the invocation, including those created outside of the program, you can call LIB$FIND_VM_ZONE. At each call, LIB$FIND_VM_ZONE scans the heap management database and returns the zone identifier of the next valid zone.

LIB$SHOW_VM_ZONE returns formatted information about a specified zone, detailing such information as the zone's name, characteristics, and areas, and then passes the information to the specified or default action routine. LIB$VERIFY_VM_ZONE verifies the zone header and scans all of the queues and lists maintained in the zone header.

If you call LIB$GET_VM to allocate memory from a zone and the zone has no free memory to satisfy the request, LIB$GET_VM calls LIB$GET_VM_PAGE to allocate a block of contiguous pages for the zone. Each such block of contiguous pages is called an area. You control the number of pages in an area by specifying the area extension size attribute when you create the zone.

The systematic use of zones provides the following benefits:

- Structuring heap memory management

  Data structures in your program may have different life spans or dynamic scopes. Some structures may continue to grow during the entire execution of your program, while others exist for a very short time and are then discarded by the program. You can create a separate zone in which you allocate a particular type of short-lived structure. When the program no longer needs any of those structures, you can delete all of them in a single operation by calling LIB$DELETE_VM_ZONE.

- Program locality

  Program locality is a characteristic of a program that indicates the distance between the references and virtual memory over a period of time. A program with a high degree of program locality does not refer to many widely scattered virtual addresses in a short period of time. Maintaining a high degree of program locality reduces the number of page faults and improves program performance.

  It is important to minimize the number of page faults to obtain best performance in a virtual memory system such as VAX and Alpha systems. For example, if your program creates and searches a symbol table, you can reduce the number of page faults incurred by the search operation by using as few pages as possible to hold all the symbol table entries. If you allocate symbol table entries and other items unrelated to the symbol table in the same zone, each page of the symbol table contains both symbol table entries and other items. Because of the extra unrelated entries, the symbol table takes up more pages than it actually needs. A search of the symbol table then accesses more pages, and performance is lower as a result. You may be able to reduce the number of page faults by creating a separate symbol table zone so that pages that contain symbol table entries do not contain any unrelated items.

- Specialized allocation algorithms

  No single memory allocation algorithm is ideal for all applications. Section 21.6 describes the run-time library memory allocation algorithms and their performance characteristics so that you can select an appropriate algorithm for each zone that you create.

- Performance tuning

  You can specify a number of attributes that affect performance when you create a zone. The allocation algorithm you select can have a significant effect on performance. By specifying the allocation block size, you can improve performance and reduce fragmentation within the zone at the cost of some extra memory. Boundary tags can also be used to improve the speed of

LIB$FREE_VM at the cost of some extra memory. Boundary tags are further discussed in Section 21.4.1.

21.4.1 Zone Attributes

You can specify a number of zone attributes when you call LIB$CREATE_VM_ZONE to create the zone. The attributes that you specify are permanent; that is, you cannot change the attribute values. They remain constant until you delete the zone. Each zone that you create can have a different set of attribute values. Thus, you can tailor each zone to optimize program locality, execution time, and memory usage.

This section describes each of the zone attributes, suggested values for the attribute, and the effects of the attribute on execution time and memory usage. If you do not specify a complete set of attribute values, LIB$CREATE_VM_ZONE provides defaults for many of the attributes. More detailed information about argument names and the encoding of arguments is given in the description of LIB$CREATE_VM_ZONE in the *OpenVMS RTL Library (LIB$) Manual*.

The zone attributes are as follows:

- Allocation algorithms

  The run-time library heap management routines provide four algorithms to allocate and free memory and to manage blocks of free memory. The algorithms are listed here. (See Section 21.6 for more details.)

  - The First Fit algorithm (LIB$K_VM_FIRST_FIT) maintains a linked list of free blocks, sorted in order of increasing memory address.
  - The Quick Fit algorithm (LIB$K_VM_QUICK_FIT) maintains a set of lookaside lists indexed by request size for request sizes in a specified range. For request sizes that are not in the specified range, a First Fit list of free blocks is maintained by the heap management routines.
  - The Frequent Sizes algorithm (LIB$K_VM_FREQ_SIZES) is similar to Quick Fit in that it maintains a set of lookaside lists for some block sizes. You specify the number of lists when you create the zone; the sizes associated with those lists are determined by the actual sizes of blocks that are freed.
  - The Fixed Size algorithm (LIB$K_VM_FIXED) maintains a single queue of free blocks.

- Boundary-tagged blocks

  You can specify the use of boundary tags (LIB$M_VM_BOUNDARY_TAGS) with any of the algorithms that handle variable-sized blocks. The algorithms that handle variable-sized blocks are First Fit, Quick Fit, and Frequent Sizes.

  If you specify boundary tags, LIB$GET_VM appends two additional longwords to each block that you allocate. LIB$FREE_VM uses these tags to speed up the process of merging adjacent free blocks on the First Fit free list. Using the standard First Fit insertion and merge, the execution time and number of page faults to free a block are proportional to the number of items on the list; freeing *n* blocks takes time proportional to *n* squared. When boundary tags are used, LIB$FREE_VM does not have to keep the free list in sorted order. This reduces the time and the number of page faults for freeing one block to a constant value that is independent of the number of free blocks. By using boundary tags, you can improve execution time at the cost of some additional memory for the tags.

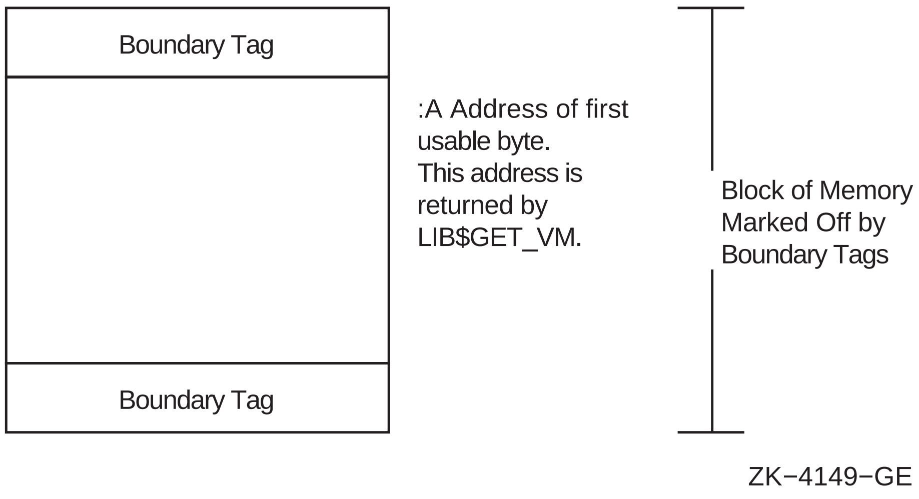
The use of boundary tags can have a significant effect on execution time if *all* of the following three conditions are present:

- You are using the First Fit algorithm.
- There are many calls to LIB$FREE_VM.
- The free list is long.

Boundary tags do not improve execution time if you are using Quick Fit or Frequent Sizes and if all the blocks being freed use one of the lookaside lists. Merging or searching is not done for free blocks on a lookaside list.

The boundary tags specify the length of each block that is allocated, so you do not need to specify the length of a tagged block when you free it. This reduces the bookkeeping that your program must perform. Figure 21–2 shows the placement of boundary tags.

Figure 21–2 Boundary Tags


Boundary Tag
:A Address of first usable byte. This address is returned by LIB$GET_VM.
Block of Memory Marked Off by Boundary Tags
Boundary Tag
ZK−4149−GE

Boundary tags are not visible to the calling program. The request size you specify when calling LIB$GET_VM is the number of usable bytes your program needs. The address returned by LIB$GET_VM is the address of the first usable byte of the block, and this same address is used when you call LIB$FREE_VM.

- Area extension size

  Pages of memory are allocated to a zone in contiguous groups called areas. By specifying area extension parameters for the zone, you can tailor the zone to achieve a satisfactory balance between locality, memory usage, and execution time for allocating pages. If you specify a large area size, you improve locality for blocks in the zone, but you may waste a large amount of virtual memory. Pages can be allocated to an area of a zone, but the memory might never be used to satisfy a LIB$GET_VM allocation request. If you specify a small area extension size, you reduce the number of pages used, but you may reduce locality and you increase the amount of overhead for area control blocks.

You can specify two area extension size values: an initial size and an extend size. If you specify an initial area size, that number of pages is allocated to the zone when you create the zone. If you do not specify an initial size, no pages are allocated until the first call to LIB$GET_VM that references the zone. When an allocation request cannot be satisfied by blocks from the free list or from memory in any of the areas owned by the zone, a new area is added to the zone. The size of this area is the maximum of the area extend size and the current request size. The extend size does not limit the size of blocks you can allocate. If you do not specify extend size when you create the zone, a default of 16 pages is used.

Choose a few area extension sizes, and use them throughout your program. It is also desirable to choose extension sizes that are multiples of each other. Memory for areas is allocated by calling LIB$GET_VM_PAGE. You should choose the area extension sizes in order to minimize fragmentation. Digital-supplied software generally uses extension sizes that are a power of 2.

Also consider the overhead for area control blocks when choosing the area extension parameters. Each area control block is 64 bytes long. Table 21–1 shows the overhead for various extension sizes.

Table 21–1 Overhead for Area Control Blocks

Area Size (Pages)	Overhead Percentage
1	12.5%
2	6.3%
4	3.1%
16	0.8%
128	0.1%

You can also control the way in which zones are extended by using the LIB$M_VM_EXTEND_AREA attribute. This attribute specifies that when new pages are allocated for a zone, they should be appended to an existing area if the pages are adjacent to an existing area.

- Block size

  The block size attribute specifies the number of bytes in the basic allocation quantum for the zone.

  All allocation requests are rounded up to a multiple of the block size.

  The block size must be a power of 2 in the range of 8 to 512. Table 21–2 lists the possible block sizes.

Table 21–2 Possible Values for the Block Size Attribute

Block Size (Power of 2)	Actual Block Size
2^3	8
2^4	16
2^5	32

(continued on next page)

Table 21–2 (Cont.) Possible Values for the Block Size Attribute

Block Size (Power of 2)	Actual Block Size
2^6	64
2^7	128
2^8	256
2^9	512

By adjusting the block size, you can control the effects of internal fragmentation and external fragmentation. Internal fragmentation occurs when the request size is rounded up and more bytes are allocated than are required to satisfy the request. External fragmentation occurs when there are many small blocks on the free list, but none of them is large enough to satisfy an allocation request.

If you do not specify a value for block size, a default of 8 bytes is used.

- Alignment

  The alignment attribute specifies the required address boundary alignment for each block allocated. The alignment value must be a power of 2 in the range of 4 to 512.

  The block size and alignment values are closely related. If you are not using boundary-tagged blocks, the larger value of block size and alignment controls both the block size and alignment. If you are using boundary-tagged blocks, you can minimize the overhead for the boundary tags by specifying a block size of 8 and an alignment of 4 (longword alignment).

  VAX

  Note that the VAX interlocked queue instructions require quadword alignment, so you should not specify longword alignment for blocks that will be inserted on an interlocked queue. ♦

  If you do not specify an alignment value, a default of 8 is used (alignment on a quadword boundary).

- Page limit

  You can specify the maximum number of pages that can be allocated to the zone. If you do not specify a limit, the only limit is the total process virtual address limit imposed by process quotas and system parameters.

- Fill on allocate

  If you do not specify the allocation fill attribute, LIB$GET_VM does not initialize the contents of the blocks of memory that it supplies. The contents of the memory are unpredictable, and you must assign a value to each location your program uses.

  In many applications, it is convenient to initialize every byte of dynamically allocated memory to the value 0. You can request that LIB$GET_VM do this initialization by specifying the allocation fill attribute LIB$M_VM_GET_FILL0 when you create the zone.

  If your program does not use the allocation fill attribute, it may be very difficult to locate bugs where the program does not properly initialize dynamically allocated memory. As a debugging aid, you can request that LIB$GET_VM initialize every byte to FF (hexadecimal) by specifying the allocation fill attribute LIB$M_VM_GET_FILL1 when you create the zone.

- Fill on free

  In complex programs using heap storage, it can be very difficult to locate bugs where the program frees a block of memory but continues to make references to that block of memory. As a debugging aid, you can request that LIB$FREE_VM write bytes containing 0 or FF (hexadecimal) into each block of memory when it is freed; specify one of the attributes LIB$M_VM_FREE_FILL0 or LIB$M_VM_FREE_FILL1.

21.4.2 Default Zone

The run-time library provides a default zone that is used if you do not specify a **zone-id** argument when you call LIB$GET_VM or LIB$FREE_VM. The default zone provides compatibility with earlier versions of LIB$GET_VM and LIB$FREE_VM, which did not support multiple zones.

Programs that do not place heavy demands on heap storage can simply use the default zone for all heap storage allocation. They do not need to call LIB$CREATE_VM_ZONE and LIB$DELETE_VM_ZONE, and they can omit the **zone-id** argument when calling LIB$GET_VM and LIB$FREE_VM. The **zone-id** for the default zone has the value 0.

The default zone has the values shown in Table 21–3.

Table 21–3 Attribute Values for the Default Zone

Attribute	Value
Algorithm	First Fit
Area extension size	128 pages
Block size	8 bytes
Alignment	Quadword boundary
Boundary tags	No boundary tags
Page limit	No limit
Fill on allocate	No fill on allocate
Fill on free	No fill on free

21.4.3 Zone Identification

A zone is a logically independent memory pool or subheap. You can create zones by calling LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE. These routines return as an output argument a unique 32-bit zone identifier (**zone-id**) that is used in subsequent routine calls where a zone identification is needed.

You can specify the **zone-id** argument as an optional argument when you call LIB$GET_VM to allocate a block of memory. If you do specify **zone-id** when you allocate memory, you must specify the same **zone-id** value when you call LIB$FREE_VM to free the memory. LIB$FREE_VM returns an error status if you do not provide the correct value for **zone-id**.

Modular routines that allocate and free heap storage must use zone identifications in a consistent fashion. You can use one of several approaches in designing a set of modular routines to ensure consistency in using zone identifications:

- Each routine that allocates or frees heap storage has a **zone-id** argument so the caller can specify the zone to be used.

- The modular routine package provides ALLOCATE and FREE routines for each type of dynamically allocated object. These routines keep track of zone identifications in an implicit argument, in static storage, or in the dynamically allocated objects. The caller need not be concerned with the details of zone identifications.
- By convention, the set of modular routines could do all allocate and free operations in the default zone.

The zone identifier for the default zone has the value 0 (see Section 21.4.2 for more information on the default zone). You can allocate and free blocks of memory in the default zone by specifying a **zone-id** value of 0 or by omitting the **zone-id** argument when you call LIB$GET_VM and LIB$FREE_VM. You cannot use LIB$DELETE_VM_ZONE or LIB$RESET_VM_ZONE on the default zone; these routines return an error status if the value for **zone-id** is 0.

21.4.4 Creating a Zone

The LIB$CREATE_VM_ZONE routine creates a new zone and sets zone attributes according to arguments that you supply. LIB$CREATE_VM_ZONE returns a **zone-id** value for the new zone that you use in subsequent calls to LIBGET_VM, LIBFREE_VM, and LIB$DELETE_VM_ZONE.

21.4.5 Deleting a Zone

The LIB$DELETE_VM_ZONE routine deletes a zone and returns all pages owned by the zone to the processwide page pool managed by LIB$GET_VM_PAGE. Your program must not perform any more operations on the zone after you call LIB$DELETE_VM_ZONE.

It takes less execution time to free memory in a single operation by calling LIB$DELETE_VM_ZONE than to account individually for and free every block of memory that was allocated by calling LIB$GET_VM. Of course, you must be sure that your program is no longer using the zone or any of the memory in the zone before you call LIB$DELETE_VM_ZONE.

If you have specified deallocation filling, LIB$DELETE_VM_ZONE fills all of the allocated blocks that are freed.

21.4.6 Resetting a Zone

The LIB$RESET_VM_ZONE routine frees all the blocks of memory that were previously allocated from the zone. The memory becomes available to satisfy further allocation requests for the zone; the memory is not returned to the processwide page pool managed by LIB$GET_VM_PAGE. Your program can continue to use the zone after you call LIB$RESET_VM_ZONE.

It takes less execution time to free memory in a single operation by calling LIB$RESET_VM_ZONE than to account individually for and free every block of memory that was allocated by calling LIB$GET_VM. Of course, you must be sure that your program is no longer using any of the memory in the zone before you call LIB$RESET_VM_ZONE.

If you have specified deallocation filling, LIB$RESET_VM_ZONE fills all of the allocated blocks that are freed.

Because LIB$RESET_VM_ZONE does not return any pages to the processwide page pool, you should reset a zone only if you expect to reallocate almost all of the memory that is currently owned by the zone. If the next cycle of reallocation may use much less memory, it is better to delete the zone (with LIB$DELETE_VM_ZONE) and create it again (with LIB$CREATE_VM_ZONE).

21.5 Allocating and Freeing Blocks

The run-time library heap management routines LIB$GET_VM and LIB$FREE_VM provide the mechanism for allocating and freeing blocks of memory.

The LIB$GET_VM and LIB$FREE_VM routines are fully reentrant, so they can be called by code running at AST level or in an Ada multitasking environment. Several threads of execution can be allocating or freeing memory simultaneously in the same zone or in different zones.

All memory allocated by LIB$GET_VM has user-mode read/write access, even if the call to LIB$GET_VM is made from a more privileged access mode.

The rules for using LIB$GET_VM and LIB$FREE_VM are as follows:

- Any memory you free by calling LIB$FREE_VM must have been allocated by a previous call to LIB$GET_VM. You cannot allocate memory by calling SYS$EXPREG or SYS$CRETVA and then free it using LIB$FREE_VM.
- When you free a block of memory by calling LIB$FREE_VM, you must use the same **zone-id** value as when you called LIB$GET_VM to allocate the block. If the block was allocated from the default zone, you must either specify a **zone-id** value of 0 or omit the **zone-id** argument when you call LIB$FREE_VM.
- You cannot free part of a block that was allocated by a call to LIB$GET_VM; the whole block must be freed by a single call to LIB$FREE_VM.
- You cannot combine contiguous blocks of memory that were allocated by several calls to LIB$GET_VM into one larger block that is freed by a single call to LIB$FREE_VM.
- All memory allocated by LIB$GET_VM is aligned according to the alignment attribute for the zone; all memory freed by LIB$FREE_VM must have the correct alignment for the zone. An error status is returned if you attempt to free a block that is not aligned properly.

21.6 Allocation Algorithms

The run-time library heap management routines provide four algorithms, listed in Table 21–4, that are used to allocate and free memory and that are used to manage blocks of free memory.

Table 21–4 Allocation Algorithms

Code	Symbol	Description
1	LIB$K_VM_FIRST_FIT	First Fit
2	LIB$K_VM_QUICK_FIT	Quick Fit (maintains lookaside list)
3	LIB$K_VM_FREQ_SIZES	Frequent Sizes (maintains lookaside list)
4	LIB$K_VM_FIXED	Fixed Size Blocks

The Quick Fit and Frequent Sizes algorithms use lookaside lists to speed allocation and freeing for certain request sizes. A lookaside list is the software analog of a hardware cache. It takes less time to allocate or free a block that is on a lookaside list.

For each of the algorithms, LIB$GET_VM performs one or more of the following operations:

- Tries to allocate a block from an appropriate lookaside list.
- Scans the list of areas owned by the zone. For each area, it tries to allocate a block from the free list and then tries to allocate a block from the block of unallocated memory at the end of the area.
- Adds a new area to the zone and allocates the block from that area.

For each of the algorithms, LIB$FREE_VM performs one or more of the following operations:

- Places the block on a lookaside list associated with the zone if there is an appropriate list.
- Locates the area that contains the block. If the zone has boundary tags, the tags encode the area; otherwise, it scans the list of areas owned by the zone to find the correct area.
- Inserts the block on the area free list and checks for merges with adjacent free blocks.

  If the zone has boundary tags, LIB$FREE_VM checks the tags of adjacent blocks; if a merge does not occur, it inserts the block at the tail of the free list.

  If the zone does not have boundary tags, LIB$FREE_VM scans the sorted free list to find the correct insertion point. It also checks the preceding and following blocks for merges.

21.6.1 First Fit Algorithm

The First Fit algorithm (LIB$K_VM_FIRST_FIT) maintains a linked list of free blocks. If the zone does not have boundary tags, the free list is kept sorted in order of increasing memory address. An allocation request is satisfied by the first block on the free list that is large enough; if the first free block is larger than the request size, it is split and the fragment is kept on the free list. When a block is freed, it is inserted in the free list at the appropriate point; adjacent free blocks are merged to form larger free blocks.

21.6.2 Quick Fit Algorithm

The Quick Fit algorithm (LIB$K_VM_QUICK_FIT) maintains a set of lookaside lists indexed by request size for request sizes in a specified range. For request sizes that are not in the specified range, a First Fit list of free blocks is maintained. An allocation request is satisfied by removing a block from the appropriate lookaside list; if the lookaside list is empty, a First Fit allocation is done. When a block is freed, it is placed on a lookaside list or the First Fit list according to its size.

Free blocks that are placed on a lookaside list are neither merged with adjacent free blocks nor split to satisfy a request for a smaller block.

21.6.3 Frequent Sizes Algorithm

The Frequent Sizes algorithm (LIB$K_VM_FREQ_SIZES) is similar to the Quick Fit algorithm in that it maintains a set of lookaside lists for some block sizes. You specify the number of lookaside lists when you create the zone; the sizes associated with those lists are determined by the actual sizes of blocks that are freed. An allocation request is satisfied by searching the lookaside lists for a matching size; if no match is found, a First Fit allocation is done. When a block is freed, the block is placed on a lookaside list with a matching size, on an empty lookaside list, or on the First Fit list if no lookaside list is available. As with the Quick Fit algorithm, free blocks on lookaside lists are not merged or split.

21.6.4 Fixed Size Algorithm

The Fixed Size algorithm (LIB$K_VM_FIXED) maintains a single queue of free blocks. There is no First Fit free list, and splitting or merging of blocks does not occur.

21.7 User-Defined Zones

When you create a zone by calling LIB$CREATE_VM_ZONE, you must select an allocation algorithm from the fixed set provided by the run-time library. You can tailor the characteristics of the zone by specifying various zone attributes. User-defined zones provide additional flexibility and control by letting you supply routines for the allocation and deallocation algorithms.

You create a user-defined zone by calling LIB$CREATE_USER_VM_ZONE. Instead of supplying values for a fixed set of zone attributes, you provide routines that perform the following operations for the zone:

- Allocate a block of memory
- Free a block of memory
- Reset the zone
- Delete the zone

Each time that one of the run-time library heap management routines (LIBGET_VM, LIBFREE_VM, LIB$RESET_VM_ZONE, LIB$DELETE_VM_ZONE) is called to perform an operation on a user-defined zone, the corresponding routine that you specified is called to perform the actual operation. You need not make any changes in the calling program to use user-defined zones; their use is transparent.

You do not need to provide routines for all four of the preceding operations if you know that your program will not perform certain operations. If you omit some of the operations and your program attempts to use them, an error status is returned.

Applications of user-defined zones include the following:

- You can provide your own specialized allocation algorithms. These algorithms can in turn invoke LIBGET_VM, LIBGET_VM_PAGE, SYS$EXPREG, or other system services.
- You can use a user-defined zone to monitor memory allocation operations. Example 21–1 shows a monitoring program that prints a record of each call to allocate or free memory in a zone.

Example 21–1 Monitoring Heap Operations with a User-Defined Zone

```
C+
C This is the main program that creates a zone and exercises it.
C
C Note that the main program simply calls LIB$GET_VM and LIB$FREE_VM.
C It contains no special coding for user-defined zones.
C-
        PROGRAM MAIN
        IMPLICIT INTEGER(A-Z)

        CALL MAKE_ZONE(ZONE)

        CALL LIB$GET_VM(10, I1, ZONE)
        CALL LIB$GET_VM(20, I2, ZONE)
        CALL LIB$FREE_VM(10, I1, ZONE)
        CALL LIB$RESET_VM_ZONE(ZONE)
        CALL LIB$DELETE_VM_ZONE(ZONE)
        END

C+
C This is the subroutine that creates a user-defined zone for monitoring.
C Each GET, FREE, or RESET prints a line of output on the terminal.
C Errors are signaled.
C-
        SUBROUTINE MAKE_ZONE(ZONE)
        IMPLICIT INTEGER (A-Z)
        EXTERNAL GET_RTN, FREE_RTN, RESET_RTN, LIB$DELETE_VM_ZONE

C+
C Create the primary zone.  The primary zone supports
C the actual allocation and freeing of memory.
C-
        STATUS = LIB$CREATE_VM_ZONE(REAL_ZONE)
        IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C+
C Create a user-defined zone that monitors operations on REAL_ZONE.
C-
        STATUS = LIB$CREATE_USER_VM_ZONE(USER_ZONE, REAL_ZONE,
        1       GET_RTN,
        1       FREE_RTN,
        1       RESET_RTN,
        1       LIB$DELETE_VM_ZONE)
        IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C+
C Return the zone-id of the user-defined zone to the caller to use.
C-
        ZONE = USER_ZONE
        END

C+
C GET routine for user-defined zone.
C-
        FUNCTION GET_RTN(SIZE, ADDR, ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$GET_VM(SIZE, ADDR, ZONE)
```

(continued on next page)

Example 21–1 (Cont.) Monitoring Heap Operations with a User-Defined Zone

```
        IF (.NOT. STATUS) THEN
                CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
                TYPE 10, SIZE, ADDR
10              FORMAT(' Allocated ',I4,' bytes at ',Z8)
        ENDIF
        GET_RTN = STATUS
        END

C+
C FREE routine for user-defined zone.
C-

        FUNCTION FREE_RTN(SIZE, ADDR, ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$FREE_VM(SIZE, ADDR, ZONE)

        IF (.NOT. STATUS) THEN
                CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
                TYPE 20, SIZE, ADDR
20              FORMAT(' Freed ',I4,' bytes at ',Z8)
        ENDIF
        FREE_RTN = STATUS
        END

C+
C RESET routine for user-defined zone.
C-

        FUNCTION RESET_RTN(ZONE)
        IMPLICIT INTEGER(A-Z)

        STATUS = LIB$RESET_VM_ZONE(ZONE)
        IF (.NOT. STATUS) THEN
                CALL LIB$SIGNAL(%VAL(STATUS))
        ELSE
                TYPE 30, ZONE
30              FORMAT(' Reset zone at ', Z8)
        ENDIF

        RESET_RTN = STATUS
        END
```

21.8 Debugging Programs That Use Virtual Memory Zones

This section discusses some methods and aids for debugging programs that use virtual memory zones. Note that this information is implementation dependent and may change at any time.

The following list offers some suggestions for discovering and tracking problems with memory zone usage:

- Run the program with both free-fill-zero and free-fill-one set. The results from both executions of the program should be the same. If the results differ, this could mean that you are referencing a zone that is already deallocated. It could also mean that, after deallocating a zone, you created a new zone at the same location, so that you now have two pointers pointing to the same zone.
- Call LIB$FIND_VM_ZONE at image termination. If a virtual memory zone is not deleted, LIB$FIND_VM_ZONE returns its zone identifier.
- Use LIB$SHOW_VM_ZONE and LIB$VERIFY_VM_ZONE to print zone information and check for errors in the internal data structures. LIB$SHOW_VM_ZONE allows you to determine whether any linkage pointers for the virtual memory zones are corrupted. LIB$VERIFY_VM_ZONE allows you to request verification of the contents of the free blocks, so that if you call LIB$VERIFY_VM_ZONE with free-fill set, you can determine whether you are writing to any deallocated zones.
- For zones created with the Fixed Size, Quick Fit, or Frequent Size algorithm, some types of errors cannot be detected. For example, in a zone that implements the Fixed Size algorithm (or in a Quick Fit or Frequent Size algorithm when the block is cached on a lookaside list), freeing a block more than once returns SS$_NORMAL, but the internal data structures are invalid. In this case, change the algorithm to First Fit. The First Fit algorithm checks whether you are freeing a block that is already on the free list and, if so, returns the error LIB$_BADBLOADR.

22 Alignment on OpenVMS VAX and Alpha Systems

This chapter describes the importance and techniques of alignment for both OpenVMS VAX and OpenVMS Alpha systems.[1] It contains the following subsections:

Section 22.1 describes alignment on OpenVMS VAX and OpenVMS Alpha systems.

Section 22.2 describes using compilers for alignment.

Section 22.3 describes using various tools to uncover unaligned data.

22.1 Alignment

Alignment is an aspect of a data item that refers to its placement in memory. The mixing of byte, word, longword, and quadword data types can lead to data that is not aligned on natural boundaries. A naturally aligned datum of size 2**N is stored in memory at a starting byte address that is a multiple of 2**N; that is, an address that has N low-order zero bits. Data is naturally aligned when its address is an integral multiple of the size of the data in bytes (for example, when the following occurs):

- A byte is aligned at any address.
- A word is aligned at any address that is a multiple of 2.
- A longword is aligned at any address that is a multiple of 4.
- A quadword is aligned at any address that is a multiple of 8.

Data that is not aligned is referred to as unaligned. Throughout this chapter, the term aligned is used instead of naturally aligned.

Table 22–1 shows examples of common data sizes, their alignment, the number of zero bits in an aligned address for that data, and a sample aligned address in hexadecimal.

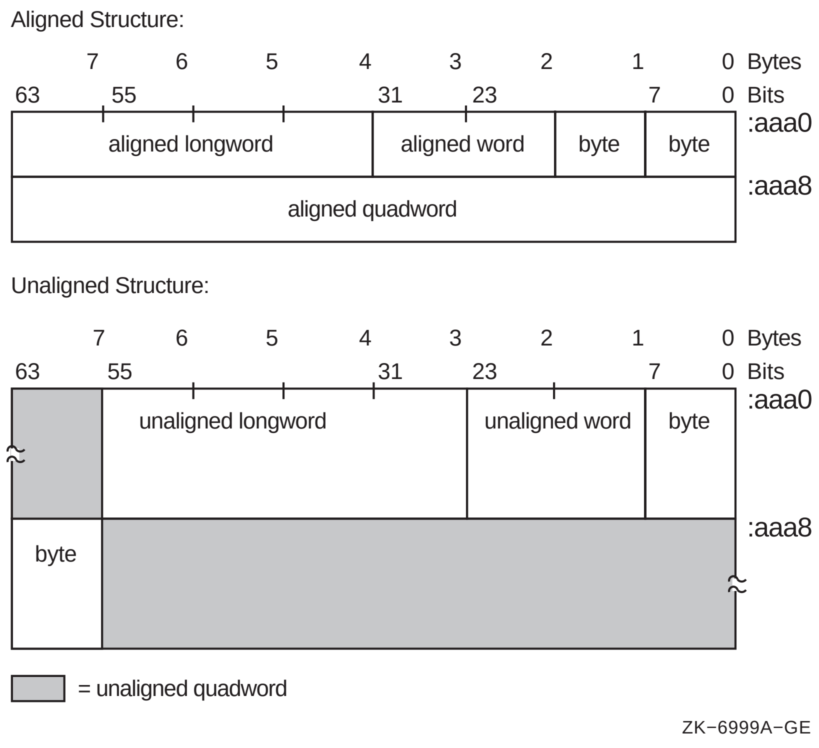
[1] Reprinted from an article in the March/April 1993 issue of *Digital Systems Journal*, Volume 15, Number 2, titled "Alpha AXP(TM) Migration: Understanding Data Alignment on OpenVMS AXP Systems" by Eric M. LaFranchi and Kathleen D. Morse. Copyright 1993 by Cardinal Business Media, Inc., 101 Witmer Road, Horsham, PA 19044.

Table 22–1 Aligned Data Sizes

Data Size	Alignment	Zero Bits	Aligned Address Example
Byte	Byte	0	10001, 10002, 10003, 10004
Word	Word	1	10002, 10004, 10006, 10008
Longword	Longword	2	10004, 10008, 1000C, 10010
Quadword	Quadword	4	10008, 10010, 10018, 10020

An aligned structure has all its members aligned. An unaligned structure has one or more unaligned members. Figure 22–1 shows examples of aligned and unaligned structures.

Figure 22–1 Aligned and Unaligned Structures



22.1.1 Alignment and Performance

To achieve optimal performance, use aligned instruction sequence references and naturally aligned data. When unaligned data is referenced, more overhead is required than when referencing aligned data. This condition is true for both OpenVMS VAX and Alpha systems. On both VAX and Alpha systems, data need not be aligned to obtain correct processing results. Alignment is a concern for performance, not program correctness. Because natural alignment is not always possible, both OpenVMS VAX and Alpha systems provide help to manage the impact of unaligned data references.

Alpha

Although alignment is not required on VAX systems for stack, data, or instruction stream references, Alpha systems require that the stack and instructions be longword aligned.♦

22.1.1.1 Alignment on OpenVMS VAX

VAX

On VAX systems, memory references that are not longword aligned result in a transparent performance degradation. The full effect of unaligned memory references is hidden by microcode, which detects the unaligned reference and generates a microtrap to handle the alignment correction. This fix of alignment is done entirely in microcode. Aligned references, on the other hand, avoid the microtraps to handle fixes. Even with this microcode fix, an unaligned reference can take up to four times longer than an aligned reference. ♦

22.1.1.2 Alignment on OpenVMS Alpha

Alpha

On Alpha systems, you can check and correct alignment the following three ways:

- Allow privileged architecture library code (PALcode) to fix the alignment faults for you.
- Use directives to the compiler.
- Fix the data yourself to make sure data is aligned.

Though Alpha systems do not use microcode to automatically handle unaligned references, PALcode traps the faults and corrects unaligned references as the data is processed. If you use the shorter load/store instruction sequences and your data in unaligned, then you incur an alignment fault PALcode fixup. The use of PALcode to correct alignment faults results in the slowest of the three ways to process your data.

By using directives to the compiler, you can tell your compiler to create a safe set of instructions. If it is unaligned, the compiler uses a set of unaligned load/store instructions. These unaligned load/store instructions are called safe sequences because they never generate unaligned data exceptions. Code sequences that use the unaligned load/store instructions are longer than the aligned load/store instruction sequences. By using unaligned load/store instructions and longer instruction sequences, you can obtain the desired results without incurring an alignment trap. This technique allows you to avoid the significant performance impact of a trap and subsequent data fixes.

By fixing the data yourself so that it is aligned, you can use a short instruction stream. This results in the fastest way to process your data. When aligning data, the following recommendations are suggested:

- If references to the data must be made atomic, then the data *must* be aligned. Otherwise, an unaligned fault causes a fatal reserved operand fault in this case.
- If you fix alignment problems in public interfaces, then you could break existing programs.

To detect unaligned reference information, you can use utilities such as the OpenVMS Debugger and Performance and Coverage Analyzer (PCA). You can also use the OpenVMS Alpha handler to generate optional informational exceptions for process space references. This allows condition handlers to track unaligned references. Alignment fault system services allow you to enable and disable the delivery of these informational exceptions. Section 22.3.3 discusses system services that you can use to report both image and systemwide alignment problems. ♦

22.2 Using Compilers for Alignment

Alpha

On Alpha systems, compilers automatically align data by default. If alignment problems are not resolved, they are at least flagged. The following sections present how the compilers for DEC C, BLISS, DEC Fortran, and MACRO-32 deal with alignment. ♦

22.2.1 The DEC C Compiler (Alpha Only)

Alpha

On Alpha systems, the DEC C compiler naturally aligns all explicitly declared data, including the elements of data structures. The pragmas **member_alignment** and **nomember_alignment** allow data structures to be aligned or packed (putting the next piece of data on the next byte boundary) in the same manner as the VAX C compiler. Additional pragmas of **member_alignment save** and **member_alignment restore** exist to save and restore the state of member alignment. These are useful to prevent alignment assumptions in one include file from affecting other source code. The following program examples show the use of these pragmas:

```
#pragma member_alignment save   1
#pragma nomember_alignment      2

struct
{
  char  byte;
  short word;
  long  longword;
} mystruct;
#pragma member_alignment restore  3
```

1 Saves the current alignment setting.

2 Sets nomember_alignment, which means that the data is to be packed in the structure mystruct.

3 Resets the alignment setting for the code that follows.

The base alignment of a data structure is set to be the alignment of the largest member in the structure. If the largest element of a data structure is a longword, for example, then the base alignment of the data structure is longword alignment.

The **malloc**() function of the DEC C Run-Time Library retrieves pointers that are at least quadword aligned. Because it is the exception rather than the rule to encounter unaligned data in C programs, the compiler assumes most data references are aligned. Pointers, for example, are always assumed to be aligned; only data structures declared with the pragma **nomember_alignment** are assumed to contain unaligned data. If the DEC C compiler believes the data might be unaligned, it generates the safe instruction sequences; that is, it uses the unaligned load/store instructions. Also, the **/WARNING=ALIGNMENT** compiler qualifier can be used to turn on alignment checking by the compiler. This results in a compiler warning for unaligned data references. ♦

22.2.1.1 Compiler Example of Memory Structure of VAX C and DEC C

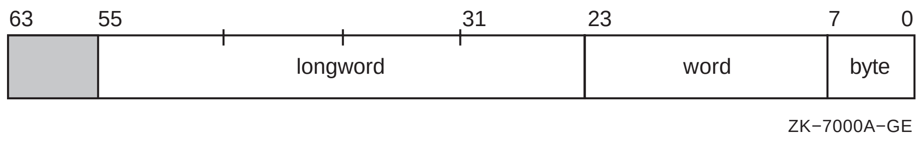
The following code examples, and Figure 22–2, and Figure 22–3 illustrate a C data structure containing byte, word, and longword data and how it would be laid out in memory by VAX C and DEC C.

```
struct
{
  char  byte;
  short word;
  long  longword;
}mystruct;
```

VAX

On VAX systems, when compiled using the VAX C compiler, the previous structure has a memory layout as shown in Figure 22–2, where each piece of data begins on the next byte boundary. ♦

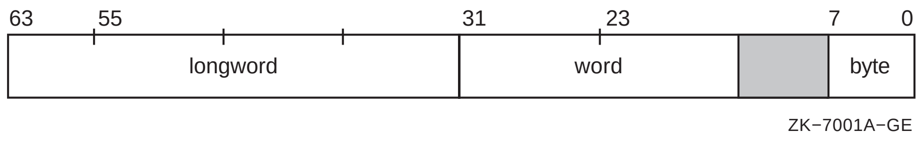
Figure 22–2 Alignment Using VAX C Compiler



Alpha

On Alpha systems, when compiled using the DEC C compiler, the structure is padded to achieve natural alignment, if needed, as shown in Figure 22–3.

Figure 22–3 Alignment Using DEC C Compiler



Note where DEC C places some padding to align naturally all the data structure elements. DEC C would also align the structure itself on a longword boundary. The DEC C compiler aligns the structure on a longword boundary because the largest element in the structure is a longword. ♦

22.2.2 The BLISS Compiler

Alpha

On Alpha systems, the BLISS compiler provide greater control over alignment than the DEC C compiler does. The BLISS compiler also makes different assumptions about alignment.

Digital does not ship the BLISS compiler on Alpha systems. ♦

The Alpha BLISS compiler, like the VAX BLISS compiler, allows explicit specification of program section (PSECT) alignment.

Alpha

On Alpha systems, BLISS compilers align all explicitly declared data on naturally aligned boundaries.

Declared data in BLISS source code can be aligned with the **ALIGN** attribute, although the alignment specified cannot be greater than that for the PSECT in which the data is contained. The alignment attribute indicates a specific address boundary by means of a boundary value, N, which specifies that the binary address of the data segment must end in at least N 0s. To specify the static byte datum A to be aligned on a longword boundary, for example, the following declaration might be used:

```
OWN
A:BYTE ALIGN(2)
```

When the BLISS compiler cannot determine the base alignment of a BLOCK, it assumes full word alignment, unless told otherwise by a command qualifier or switch declaration. Like the DEC C compiler, if the BLISS compilers believe that the data is unaligned, they generate safe instruction sequences. If you specify the qualifier **/CHECK=ALIGNMENT** in the BLISS command line, then warning information is provided when they detect unaligned memory references. ♦

22.2.3 The DEC Fortran Compiler (Alpha Only)

Alpha

On Alpha systems, the defaults for the DEC Fortran compiler emphasize compatibility and standards conformance. Normal data declarations (data declared outside of COMMON block statements) are aligned on natural boundaries by default. COMMON block statement data is not aligned by default, which conforms to the FORTRAN–77 and FORTRAN-90 standards.

The qualifier **/ALIGN=(COMMONS=STANDARD)** causes COMMON block data to be longword aligned. This adheres with the FORTRAN–77 and FORTRAN-90 standards, which state that the compiler is not allowed to put padding between INTEGER*4 and REAL*8 data. This can cause REAL*8 data to be unaligned. To correct this, apply the NATURAL rule; for instance, apply **/ALIGN=(COMMONS=NATURAL)** to get natural alignment up to quadwords and the best performance, though this is not standards conforming.

To pack COMMON block and RECORD statement data, specify **/ALIGN=NONE**. The qualifier **/ALIGN=NONE** is equivalent to **/NOALIGN**, **/ALIGN=PACKED**, or **/ALIGN=(COMMON=PACKED,RECORD=PACKED)**. To pack just RECORD statement data, specify **/ALIGN=(RECORD=PACKED)**.

Besides command line qualifiers, DEC Fortran provides two directives to control the alignment of RECORD statement data and COMMON block data. The **CDEC$OPTIONS** directive controls whether the DEC Fortran compiler naturally aligns fields in RECORD statements or data items in COMMON blocks for performance reasons, or whether the compiler packs those fields and data items together on arbitrary byte boundaries. The **CDEC$OPTIONS** directive, like the **/ALIGN** command qualifier, takes class and rule parameters. Also, the **CDE$OPTIONS** directive overrides the compiler option **/ALIGN**.

By default, the DEC Fortran compiler emits alignment warnings, but these can be turned off by using the qualifier **/WARNINGS=NOALIGNMENT**. ♦

22.2.4 The MACRO-32 Compiler (Alpha Only)

Alpha

As with the C, BLISS, and DEC Fortran languages, unaligned data references in MACRO-32 code work correctly, though they perform slower than aligned references. The MACRO-32 language provides you with direct control over alignment. There is no implicit padding for alignment done by the MACRO-32 compiler; data remains at the alignment you specify.

The MACRO-32 compiler recognizes the alignment of all locally declared data and flags all references to declared data that is unaligned. By default, the MACRO-32 compiler assumes that addresses in registers used as base pointers are longword aligned at routine entry.

For the **MOVQ** instruction, the compiler assumes that the base address is longword aligned, unless the compiler determines by its register-tracking logic that the address can not be longword aligned. Only longword alignment is tracked; quadword register alignment is not tracked. If you use the OpenVMS Alpha **MOVQ** instruction, the data must be quadword aligned, because the compiler does not translate it into unaligned loads.

External data is data that is not contained in the current source being compiled. External data is assumed to be longword aligned by the MACRO-32 compiler. The compiler detects and flags unaligned global label references. This enables you to locate external data that is not aligned.

To preserve atomicity, the compiler assumes that the data is longword aligned. Unaligned data causes a trap and voids the atomicity. Therefore, you must ensure that such data is aligned.

To fix unaligned data references, the easiest way is for you to align the data, if possible. If you cannot align the data, the data address can be moved into a register and then the register declared as unaligned. When you compile with **/UNALIGNED**, you tell the compiler to treat all data references as unaligned and to generate safe unaligned sequences. You can also use the **.SET_REGISTERS** directive, which affects data references only for the specified registers for a section of code.

The **.PSECT** and **.ALIGN** directives are supported. The compiler knows the alignment of locally declared data. The compiler makes certain assumptions about the alignment, but does allow programmer control over those assumptions. The MACRO-32 compiler provides two directives for changing the compiler's assumptions about alignment, which results in letting the compiler produce more efficient code. These two directives are as follows:

- **.SET_REGISTERS** allows you to specify whether a register points to aligned or unaligned data. You use this directive when the result of an operation is the opposite of what the compiler expects. Also, use the same directive to declare registers that the compiler would not otherwise detect as input or output registers.

  For example, consider the **DIVL** instruction. After executing this instruction in the following example, the MACRO-32 compiler assumes that R1 is unaligned. A future attempt at using R1 as a base register will cause the compiler to generate an unaligned fetch sequence. However, suppose you know that R1 is always aligned after the **DIVL** instruction. You can then use the **.SET_REGISTERS** directive to inform the compiler of this. When the compiler sees the **MOVL** from 8(r1), it knows that it can use the shorter aligned fetch (LDL) to retrieve the data. At run time, however, if R1 is not really aligned, then this results in an alignment trap. The following example show the setting of a register to be aligned:

```
divl   r0,r1            ;Compiler now thinks R1 unaligned
.set_registers aligned=r1
movl   8(r1),r2         ;Compiler now treats R1 as aligned
```

- **.SYMBOL_ALIGNMENT** allows you to specify the alignment of any memory reference that uses a symbolic offset. The **.SYMBOL_ALIGNMENT** directive associates an alignment attribute with a symbol definition used as a register offset; it can be used when you know the base register will be aligned for every use of the symbolic offset. This attribute guarantees to the compiler that the base register has that alignment, and this enables the compiler to generate optimal code.

  In the example that follows, **QUAD_OFF** has a symbol alignment of **QUAD, LONG_OFF**, a symbol alignment of **LONG**, and **NONE_OFF** has no symbol alignment. In the first **MOVL** instruction, the compiler assumes that R0, since it is used as a base register with **QUAD_OFF**, is quadword aligned. Since **QUAD_OFF** has a value of 4, the compiler knows it can generate an aligned longword fetch. For the second **MOVL**, R0 is assumed to be longword aligned, but since **LONG_OFF** has a value of 5, the compiler realizes that offset+ base is not longword aligned and would generate a safe unaligned fetch sequence. In the third **MOVL**, R0 is assumed to be unaligned, unless the compiler knows otherwise from other register tracking, and would generate a safe unaligned sequence. The **.SYMBOL_ALIGNMENT** alignment remains in effect until the next occurrence of the directive.

```
.symbol_alignment QUAD
 quad_off=4
.symbol_alignment LONG
long_off=5
.symbol_alignment NONE
none_off=6

movl quad_off(r0),r1    ;Assumes R0 quadword aligned
movl long_off(r0),r2    ;Assumes R0 longword aligned
movl none_off(r0),r3    ;No presumed alignment for R0
```

  ♦

22.2.5 The VAX Environment Software Translator—VEST (Alpha Only)

Alpha

The DECmigrate for OpenVMS Alpha VAX Environment Software Translator (VEST) utility is a tool that translates binary OpenVMS VAX image files into OpenVMS Alpha image files. Image files are also called executable files. Though it is similar to compiler, VEST is for binaries instead of sources.

VEST deals with alignment in two different modes: pessimistic and optimistic. VEST is optimistic by default; but whether optimistic or pessimistic, the alignment of program counter (PC) relative data is known at translation time, and the appropriate instruction sequence can be generated.

In pessimistic mode, all non PC-relative references are treated as unaligned using the safe access sequences. In optimistic mode, the emulated VAX registers (R0–R14) are assumed to be quadword aligned upon entry to each basic block. Autoincrement and autodecrement changes to the base registers are tracked. The offset plus the base register alignment are used to determine the alignment and the appropriate access sequence is generated.

The **/OPTIMIZE=NOALIGN** qualifier on the VEST command tells VEST to be pessimistic; it assumes that base registers are not aligned, and should generate the safe instruction sequence. Doing this can slow execution speed by a factor of two or more, if there are no unaligned data references. On the other hand, it can result in a performance gain if there are a significant number of unaligned references, since safe sequences avoid any unaligned data traps.

There exist additional controls to preserve atomicity in longword data that is not naturally aligned. Wherever possible, data should be aligned in the VAX source code and the image rebuilt before translating the image with DECmigrate. This results in better performance on both VAX and Alpha systems. ♦

22.3 Using Tools for Finding Unaligned Data

Tools that aid the uncovering of unaligned data include the OpenVMS Debugger, Performance and Coverage Analyzer (PCA), and eight system services. These tools are discussed in the following sections.

22.3.1 The OpenVMS Debugger

By using the OpenVMS Debugger, you can turn on and off unaligned data exception breakpoints by using the commands **SET BREAK/UNALIGNED_DATA** and **CANCEL BREAK/UNALIGNED_DATA**. These commands must be used with the **SET BREAK/EXCEPTION** command. When the debugger breaks at the unaligned data exception, the context is like any other exception. You can examine the program counter (PC), processor status (PS), and virtual address of the unaligned data exception. Example 22–1 shows the output from the debugger using the **SET OUTPUT LOG** command of a simple program.

Example 22–1 OpenVMS Debugger Output from SET OUTPUT LOG Command

```
#include <stdio.h>
#include <stdlib.h>

main( )
{
     char *p;
     long *lp;

          /* malloc returns at least quadword aligned printer */
     p = (char *)malloc( 32 );

          /* construct unaligned longword pointer and place into lp */
     lp = (long *)((char *)(p+1));

          /* load data into unaligned longword */
     lp[0] = 123456;

     printf( "data - %d\n", lp[0] );
     return;
}
```

(continued on next page)

Example 22–1 (Cont.) OpenVMS Debugger Output from SET OUTPUT LOG Command

```
------- Compile and Link commands -------
$ cc/debug debug_example
$ link/debug debug_example
$ run debug_example
------- DEBUG session using set output log -------
Go
! break at routine DEBUG_EXAMPLE\main
!     598:          p - (char *)malloc( 32 );
set break/unaligned_data
set break/exception
set radix hexadecimal
Go
!Unaligned data access: virtual address - 003CEEA1, PC - 00020048
!break on unaligned data trap preceding DEBUG_EXAMPLE\main\%LINE 602
!     602:          printf( "data - %d\n", lp[0] );
ex/inst 00020048-4
!DEBUG_EXAMPLE\main\%LINE 600+4:                STL         R1,(R0)
ex r0
!DEBUG_EXAMPLE\main\%R0: 00000000 003CEEA1
```

22.3.2 The Performance and Coverage Analyzer—PCA

The PCA allows you to detect and fix performance problems. Because unaligned data handling can significantly increase overhead, PCA has the capability to collect and present information on aligned data exceptions. PCA commands that collect and display unaligned data exceptions are:

- **SET UNALIGNED_DATA**
- **PLOT/UNALIGNED_DATA PROGRAM BY LINE**

Also, PCA can display data according to the PC of the fault, or by the virtual address of the unaligned data.

22.3.3 System Services (Alpha Only)

Alpha

There are eight system services to help locate unaligned data. The first three system services establish temporary image reporting; the next two provide process-permanent reporting; and the last three provide for system alignment fault tracking. The symbols used in calling all eight of these system services are located in $AFRDEF in the OpenVMS Alpha MACRO-32 library, SYS$LIBRARY:STARLET.MLB. You can also call these system services in C with #include <afrdef.h>.

The first three system services can be used together; they report on the currently executing image. They are as follows:

- SYS$START_ALIGN_FAULT_REPORT. This service enables unaligned data exception for the current image. You can use either a buffered or an exception method of reporting, but you can enable only one method at a time.
  - Buffered method. This method requires that the buffer address and size be specified. You use the SYS$GET_ALIGN_FAULT_DATA service to retrieve buffered alignment data under program control.

  - Exception method. This method requires no buffer. Unaligned data exceptions are signaled to the image, at which point a user-written condition handler takes whatever action is desired. If no user-written handler is set up, then an informational exception message is broadcast for each unaligned data trap, and the program continues to execute.

- SYS$STOP_ALIGN_FAULT_REPORT. This service cancels unaligned data exception reporting for the current image if it were previously enabled. If you do not explicitly call this routine, then reporting is disabled by the operating systems' image rundown logic.
- SYS$GET_ALIGN_FAULT_DATA. This service retrieves the accumulated, buffered alignment data when using the buffered collection method.

You can use two of the eight system services to report unaligned data exceptions for the current process. The two services are as follows:

- SYS$PERM_REPORT_ALIGN_FAULT. This service enables unaligned data exception reporting for the process. Once you enable this service, the reporting remains in effect for the process until you explicitly disable it. Once enabled, the SS$_ALIGN condition is signaled for all unaligned data exceptions while the process is active. By default, if no user-written exception handler handles the condition, this results in an information display message for each unaligned data exception.

  This service provides a convenient way of running a number of images without modifying the code in each image, and also of recording the unaligned data exception behavior of each image.
- SYS$PERM_DIS_ALIGN_FAULT_REPORT. This service disables unaligned data exception reporting for the process.

The three system services that allow you to track systemwide alignment faults are as follows:

- SYS$INIT_SYS_ALIGN_FAULT_REPORT. This service initializes system process alignment fault reporting.
- SYS$STOP_SYS_ALIGN_FAULT_REPORT. This service disables systemwide alignment fault reporting.
- SYS$GET_SYS_ALIGN_FAULT_DATA. This service obtains data from the system alignment fault buffer.

These services require CMKRNL privilege. Alignment faults for all modes and all addresses can be reported using these services. The user can also set up masks to report only certain types of alignment faults. For example, you can get reports on only kernel modes, only user PC, or only data in system space.

23 System Security Services

This chapter describes the security system services that provide various mechanisms to enhance the security of operating systems. It contains the following sections:

Section 23.1 provides an overview of the protection scheme.

Section 23.2 describes identifiers and how they are used in security.

Section 23.3 describes the rights database.

Section 23.4 describes how to create, translate, and maintain access control entries (ACEs).

Section 23.5 describes protected subsystems.

Section 23.6 describes security auditing.

Section 23.7 describes how to determine a user's access to an object.

Section 23.8 describes SYS$CHECK_PRIVILEGE system service.

Section 23.9 describes how to implement site-specific security policies.

23.1 Overview of the Operating System's Protection Scheme

The basis of the security scheme is an **identifier**, which is a 32-bit binary value that represents a set of users to the system. An identifier can represent an individual user, a group of users, or some aspect of the environment in which a user is operating. A process is a **holder** of an identifier when that identifier can represent that process to the system. The protection scheme also includes the user identification code (UIC), the authorization database, and access control lists.

Authorization Database

The authorization database consists of the system authorization file (SYSUAF.DAT), the network proxy database, and the rights list database (RIGHTSLISTS.DAT). Note that the network proxy database is called NETPROXY.DAT on Alpha systems and NET$PROXY.DAT on VAX systems. (The file NETPROXY.DAT on VAX systems is maintained for platform compatibility, translation of DECnet Phase IV node names, and layered product support.) The system **rights database** is an indexed file consisting of identifier and holder records. These records define the identifiers and the holders of those identifiers on a system. When a user logs in to the system, a process is created and LOGINOUT creates a rights list for the process from the applicable entries in the rights database. The **process rights list** contains all the identifiers that the process holds. A process can be the holder of a number of identifiers. These identifiers determine the access rights of the list holder. The process rights list becomes part of the process and is propagated to any created subprocesses.

Access Protection

When a process without special privileges attempts to access an object (protected by an ACL) in the system, the operating system uses the rights list when performing a protection check. The system compares the identifiers in the rights list to the protection attributes of the object and grants or denies access to the object based on the comparison. In other words, the entries in the rights list do not specifically grant access; instead, the system uses them to perform a protection check when the process attempts to access an object.

Access Control Lists

The protection scheme provides security with the mechanism of the access control list (ACL). An ACL consists of access control entries (ACEs) that specify the type of access an identifier has to an object like a file, device, or queue. When a process attempts to access an object with an associated ACL, the system grants or denies access based on whether an exact match for the identifier in the ACL exists in the process rights list.

The following sections describe each of the components of the security scheme—identifiers, rights database, process and system rights lists, protection codes, and ACLs—and the system services affecting those components.

23.2 Identifiers

The basic component of the protection scheme is an identifier. An **identifier** represents various types of agents using the system. The types of agents represented include individual users, groups of users, and environments in which a process is operating. Identifiers and their attributes apply to both processes and objects. An **identifier name** consists of 1 to 31 alphanumeric characters with no embedded blanks and must contain at least one nonnumeric character. It can include the uppercase letters A through Z, dollar signs ($), and underscores (_), as well as the numbers 0 through 9. Any lowercase letters are automatically converted to uppercase.

23.2.1 Identifier Format

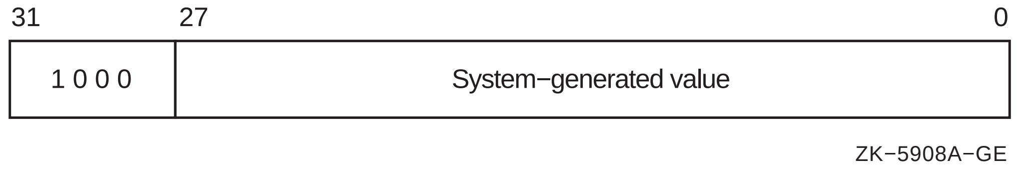
Each of the three types of identifier has an internal format in the rights database: user identification code (UIC) format, identification (ID) format, and facility-specific format. The high-order bits <31:28> of the identifier value specify the format of the identifier.

23.2.2 General Identifiers

You can define general identifiers to meet the specific needs of your site. You grant these identifiers to users by establishing holder records in the rights database. General identifiers can identify a single user, a single UIC group, a group of users, or a number of groups.

Bit <31>, which is set to 1, specifies ID format used by general identifiers as shown in Figure 23–1. Bits <30:28> are reserved by Digital. The remaining bits specify the identifier value.

Figure 23–1 ID Format



You define identifiers and their holders in the rights database with the Authorize utility or with the appropriate system services. Each user can hold multiple identifiers. This allows you to create a different kind of group designation from the one used with the user's UIC.

The alternative grouping described here permits each user to be a member of multiple overlapping groups. Access control lists (ACLs) define the access to protected objects based on the identifiers the user holds rather than on the user's UIC. See Section 23.4.3.1 for information on creating ACLs.

You can also define identifiers to represent particular terminals, times of day, or other site-specific environmental attributes. These identifiers are not given holder records in the rights database but may be granted to users by customer-written privileged software. This feature of the security system allows each site flexibility and, because the identifiers can be specific to the site, enhanced security. For a programming example demonstrating this technique, see Section 23.3.2.4. For more information, also see the *OpenVMS Guide to System Security*.

23.2.3 System-Defined Identifiers

System-defined identifiers, or **environmental** identifiers, are general identifiers that are automatically defined when the rights database is initialized. The following system-defined identifiers correspond directly with the login classes and relate to the environment in which the process operates:

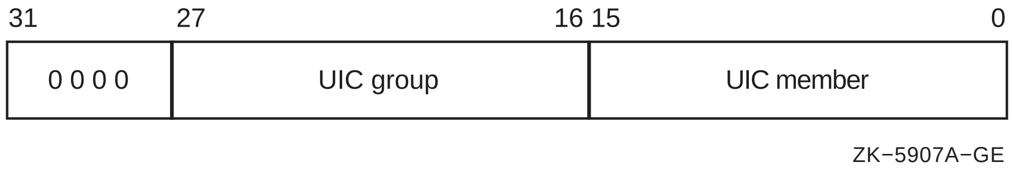
BATCH	All attempts at access made by batch jobs
NETWORK	All attempts at access made across the network
INTERACTIVE	All attempts at access made by interactive processes
LOCAL	All attempts at access made by users logged in at local terminals
DIALUP	All attempts at access made by users logged in at dialup terminals
REMOTE	All attempts at access made by users logged in on a network

Depending on the environment in which the process is operating, the system includes one or more of these identifiers when creating the process rights list.

23.2.4 UIC Identifiers

Each UIC identifier is unique and represents a system user. By default, when an account is created, its UIC is associated with the account's user name generating an identifier value. When the high-order bit <31> of the identifier value is zero, the value identifies a UIC format identifier as shown in Figure 23–2.

Figure 23–2 UIC Identifier Format



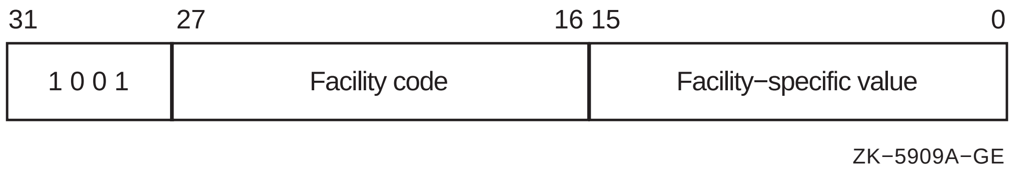
Bits <27:16> and <15:0> designate a group field and member field. Group numbers range from 1 through 16,382; member numbers range from 0 through 65,534.

23.2.5 Facility Identifiers

Facility-specific rights identifiers allow a range of unique binary identifier values to be reserved for a particular software product or application. Compare the format of facility-specific identifiers with the format of general identifiers and UIC identifiers, as shown in Section 23.2.1. The system normally determines the binary values of general identifiers when the system manager creates them; the system manager determines the binary values of UIC identifiers.

Figure 23–3 shows the facility-specific identifiers.

Figure 23–3 Facility-Specific Identifiers



The binary value of a facility-specific identifier is determined at the time the application is designed. The facility number of the identifier must match the facility number the application has chosen for its condition and message codes. The remaining 16-bit facility-specific value may be assigned at will by the application designer. By reserving specific binary identifier values, the application designer may code fixed identifier values into an application's calls to $CHECK_PRIVILEGE, $GRANT_ID, and so forth. It avoids the added complexity of first having to translate an identifier name to binary with $ASCTOID.

An application can choose to register the identifiers in the rights database or not, depending on its needs. If the identifiers are registered, they are visible to the system manager who may grant them to users. In any case, they will be displayed properly if they appear on access control lists. If they are not registered, they will remain invisible to the system manager. Unregistered identifiers that appear on access control lists are displayed as a hexadecimal value.

To register its identifiers, the installation procedure of the application must run a program that enters the identifiers into the rights database using the $ADD_IDENT service. You cannot specify facility-specific identifier values to AUTHORIZE with the ADD/IDENTIFIER command.

Typically, facility-specific identifiers serve to extend the OpenVMS privilege mechanism for an application. For example, consider a database manager that includes a function to allow appropriately privileged users to modify a schema. Access to this function could be controlled through a facility-specific identifier named, for example, DBM$MOD_SCHEMA. The system manager grants the identifier to authorized persons using the AUTHORIZE command GRANT/ID. The database services that modify schemas use the $CHECK_PRIVILEGE service to check that the caller holds the identifier.

In another example, a privileged program run by users when they log in uses $GRANT_ID to grant the user certain facility-specific identifiers, depending on conditions determined by the program; for example, time of day or access port name. These identifiers can be placed on the ACLs of files to control file access, or they might be checked by other software with $CHECK_PRIVILEGE.

23.2.6 Identifier Attributes

An identifier has attributes associated with it in the rights database. The process rights list includes the attributes of any identifiers that the process holds.

The use of rights identifiers can be extended with the following identifier attribute keywords:

DYNAMIC	Allows unprivileged holders of an identifier to add or remove the identifier from the process rights list using the DCL SET RIGHTS command. Conversely, an unprivileged user who does not have the attribute cannot modify the identifier.
HOLDER_HIDDEN	Prevents someone from using the SYS$FIND_HOLDER system service to get a list of users who hold an identifier, unless that person holds the identifier.
NAME_HIDDEN	Allows only the holders of an identifier to have it translated, either from binary to ASCII or from ASCII to binary.
NO_ACCESS	Specifies that the identifier does not affect the access rights of the user holding the identifier.
RESOURCE	Allows the holder of an identifier to charge resources, such as disk blocks, to an identifier.
SUBSYSTEM	Allows holders of the identifier to create and maintain protected subsystems.

Using the Resource Attribute

The following example demonstrates the advantages of defining an identifier and holders for a project.

The Physics department of a school has a common library with an associated disk quota on the system. In order to use the Resource attribute, you must enable disk quotas and establish a quota file entry using the SYSMAN utility. You want to allow the faculty members to charge disk quota that they use in conjunction with the library to the identifier PHYSICS associated with the common library and to prevent the students from charging resources to that identifier.

- Define an identifier PHYSICS with the Resource attribute in the rights database using the SYS$ADD_IDENT service.
- Enable disk quotas using SYSMAN as shown in the example.

```
$ MCR SYSMAN
SYSMAN> DISKQUOTA CREATE/DEVICE=DKB0:
SYSMAN> DISKQUOTA MODIFY/DEVICE=DKB0: PHYSICS /PERMQUOTA=150000 -
_SYSMAN> /OVERDRAFT=5000
SYSMAN> EXIT
```

- Create the common library and assign the identifier PHYSICS using the run-time library routine LIB$CREATE_DIR.
- Grant the identifier PHYSICS to holders FRED, a faculty member, and GEORGE, a student using the SYS$ADD_HOLDER service.

If you specify the Resource attribute for identifier FRED, he can charge disk resources to the PHYSICS identifier; if you do not specify the Resource attribute for identifier GEORGE, he will not inherit the Resource attribute associated with the identifier PHYSICS and cannot charge disk resources to the PHYSICS identifier. The following input file, USERLIST.DAT, contains valid UIC identifiers of students and faculty members:

```
FRED NORESOURCE
GEORGE RESOURCE
NANCY NORESOURCE
HAROLD RESOURCE
SUSAN RESOURCE
CHERYL NORESOURCE
MARVIN NORESOURCE
```

The following program reads USERLIST.DAT and associates the UIC identifiers with the identifier PHYSICS:

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <lib$routines.h>
#include <kgbdef.h>
#include <nam.h>
#include <string.h>
#include <stdlib.h>

#define IDENT_LEN 31
#define NO_ATTR 0

#define RESOURCE 1
#define NORESOURCE 0

unsigned int sys$asctoid(),
            sys$add_ident(),
            sys$add_holder(),
            sys$idtoasc(),
            convert_id( struct dsc$descriptor_s, unsigned int );

void add_holder( unsigned int, unsigned int, unsigned int);

struct {
    unsigned int uic;
    unsigned int terminator;
}holder;

static char ascii_ident[IDENT_LEN],
            abuffer[IDENT_LEN],
            dirbuf[NAM$C_MAXRSS],
            targbuf[IDENT_LEN];

$DESCRIPTOR(target,targbuf);

unsigned int status;

main() {
    FILE *ptr;
    char attr[11];
    unsigned int  owner_uic, attrib, resid, bin_id;
    $DESCRIPTOR(dirspec,dirbuf);
    $DESCRIPTOR(aident, abuffer);

    printf("\nEnter directory spec: ");
    gets(dirbuf);
    dirspec.dsc$w_length = strlen(dirbuf);

    printf("\nEnter its owner identifier: ");
    gets(targbuf);
    target.dsc$w_length = strlen(targbuf);


/* Add target identifier WITH resource attribute to the rights database */

    attrib = KGB$M_RESOURCE;
    status = sys$add_ident( &target, 0, attrib, &resid);
    if((status & 1) != SS$_NORMAL)
        lib$signal( status );
    else
        printf("\nAdding identifier %s to rights database...\n",
               target.dsc$a_pointer);

/* Create the common directory with the target id as owner */

    owner_uic = resid;
     status = lib$create_dir( &dirspec, &owner_uic, 0, 0);
    if((status & 1) != SS$_NORMAL)
        lib$signal( status );
    else
        printf("Creating the directory %s...\n",dirspec.dsc$a_pointer);

/* Open an input file of UIC identifiers and attribute types */
    if((ptr = fopen("USERLIST.DAT","r")) == NULL) {
        perror("OPEN");
        exit(EXIT_FAILURE);
    }

/* Read the input file of UIC identifiers */
    while((fscanf(ptr,"%s %s\n",abuffer,attr)) != EOF) {
        aident.dsc$w_length = strlen(abuffer);
        attrib = (strcmp(attr,"RESOURCE")) == 0 ? KGB$M_RESOURCE : NO_ATTR;
        bin_id = convert_id( aident, attrib);
        add_holder( bin_id, resid, attrib );
    }

/* Close the input file */
    fclose(ptr);
 }

unsigned int convert_id( struct dsc$descriptor_s uic_id,
                    unsigned int attr ) {

    unsigned int bin_id;

    status = sys$asctoid(&uic_id, &bin_id, &attr);
    if((status & 1) != SS$_NORMAL)
        lib$signal( status );
    else {
        printf("Converting identifier %s to binary format...\n",
               uic_id.dsc$a_pointer);
        return bin_id;
    }
}


void add_holder( unsigned int bin_id, unsigned int resid,
                 unsigned int attrib ) {
```

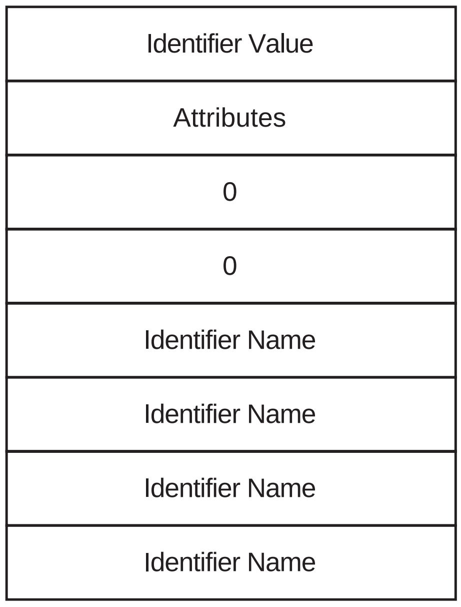
```
    int i;
    $DESCRIPTOR(nambuf,    ascii_ident);

    holder.uic = bin_id;
    holder.terminator = 0;

    status = sys$add_holder( resid, &holder, attrib);
    if((status & 1) != SS$_NORMAL)
        lib$signal( status );
    else {
        status = sys$idtoasc(bin_id, 0, &nambuf, 0, 0, 0);
        if((status & 1) != SS$_NORMAL)
            lib$signal( status );
/* Remove padding */
        nambuf.dsc$w_length = strlen(ascii_ident);
        for(i=0;i < nambuf.dsc$w_length + 1; i++)
            if (ascii_ident[i] == 0x20)
                ascii_ident[i] = '\0';
        printf("\nAdding holder %s to target identifier %s...\n", \
                        nambuf.dsc$a_pointer,target.dsc$a_pointer);
    }
}
```

23.3 Rights Database

The rights database is an indexed file containing two types of records that define all identifiers: identifier records and holder records.

One identifier record appears in the rights database for each identifier. The identifier record associates the identifier name with its 32-bit binary value and specifies the attributes of the identifier. Figure 23–4 depicts the format of the identifier record.

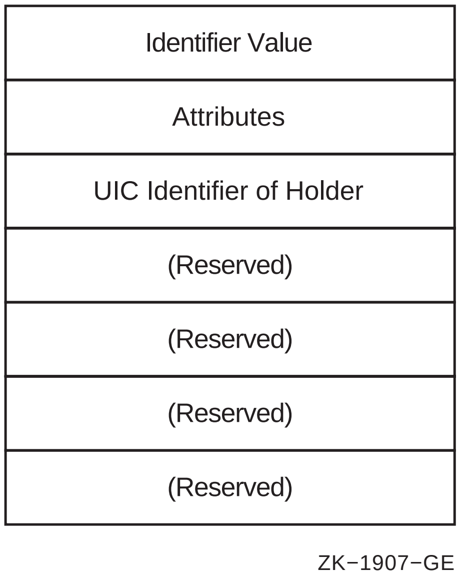
Figure 23–4 Format of the Identifier Record



ZK−1904−GE

One holder record exists in the rights database for each holder of each identifier. The holder record associates the holder with the identifier, specifies the attributes of the holder, and identifies the UIC identifier of the holder. Figure 23–5 depicts the format of the holder record.

Figure 23–5 Format of the Holder Record



The rights database is an indexed file with three keys. The primary key is the identifier value, the secondary key is the holder ID, and the tertiary key is the identifier name. Through the use of the secondary key of the holder ID, all the identifiers held by a process can be retrieved quickly when the system creates the process's rights list.

23.3.1 Initializing a Rights Database

You initialize the rights database in one of the following ways:

- When a system is installed
- With the Authorize utility
- With the SYS$CREATE_RDB system service

When you call SYS$CREATE_RDB, you can use the **sysid** argument to pass the system identification value associated with the rights database. If you omit **sysid**, the system uses the current system time in 64-bit format. If the rights database already exists, SYS$CREATE_RDB fails with the error code RMS$_FEX. To create a new rights database when one already exists, you must explicitly delete or rename the old one.

You can specify the location and name of the rights database by defining the logical name RIGHTSLIST as a system logical name in executive mode; its equivalence string must contain the device, directory, and file name of the rights database.

The file RIGHTSLIST.DAT has the protection of (S:RWED,O:RWED,G:R,W).

In order to use SYS$CREATE_RDB, write access to the database is necessary. If the database is in SYS$SYSTEM, which is the default, you need the SYSPRV privilege to grant write access to the directory.

When SYS$CREATE_RDB initializes a rights database, system-defined identifiers, which describe the environment in which a process can operate, are automatically created.

To add any other identifiers to the rights database, you must define them with the Authorize utility or with the appropriate system service.

23.3.2 Using System Services to Affect a Rights Database

The identifier and holder records in the rights database contain the following elements:

- Identifier binary value
- Identifier name
- Holders of each identifier
- Attribute of each identifier and each holder of each identifier

You can use the Authorize utility or one of the system services described in Table 23–1 to add, delete, display, modify, or translate the various elements of the rights database.

Table 23–1 Using System Services to Manipulate Elements of the Rights Database

Action	Element	Service Used
Translate	Identifier name to identifier binary value	SYS$ASCTOID
	Identifier binary value to identifier name	SYS$IDTOASC
Add	Identifier holder record	SYS$ADD_HOLDER
	New identifier record	SYS$ADD_IDENT
Find	Identifier value held by holder	SYS$FIND_HELD
	Holders of an identifier	SYS$FIND_HOLDER
	All identifiers	SYS$IDTOASC
Modify	Attribute in holder record	SYS$MOD_HOLDER
	Attribute in identifier record	SYS$MOD_IDENT
Delete	Holder from identifier record	SYS$REM_HOLDER
	Identifier and all its holders	SYS$REM_IDENT

The following table shows what access you need for which services:

Service	Required Access
SYS$ADD_HOLDER	Write
SYS$ADD_IDENT	Write
SYS$ASCTOID	Read†

†On VAX systems, read access is required when certain restrictions are present (for example, if the identifiers have the name hidden or holder hidden attributes).

Service	Required Access
SYS$CREATE_RDB	Write[1]
SYS$FIND_HELD	Read†
SYS$FIND_HOLDER	Read†
SYS$FINISH_RDB	Read†
SYS$IDTOASC	Read†
SYS$MOD_HOLDER	Write
SYS$MOD_IDENT	Write
SYS$REM_HOLDER	Write
SYS$REM_IDENT	Write

[1]File creation access.
†On VAX systems, read access is required when certain restrictions are present (for example, if the identifiers have the name hidden or holder hidden attributes).

23.3.2.1 Translating Identifier Values and Identifier Names

To the system, an identifier is a 32-bit binary value; however, to make identifiers easy to use, each binary value has an associated identifier name. The identifier value and the ASCII identifier name string are associated in the rights database. You can use the SYS$ASCTOID and SYS$IDTOASC system services to translate from one format to another. When you pass to SYS$ASCTOID the address of a string descriptor pointing to an identifier name, the corresponding identifier binary value is returned. Conversely, you use the SYS$IDTOASC service to translate a binary identifier value to an ASCII identifier name string.

Preventing a Translation

You can prevent a translation operation by unauthorized users by specifying the KGB$V_NAME_HIDDEN within an attributes mask.

Listing Identifiers in the Rights Database

You can also use the SYS$IDTOASC service to list the identifier names of all of the identifiers in the rights database. Specify the **id** argument as −1, initialize the **context** argument to 0, and repeatedly call SYS$IDTOASC until the status code SS$_NOSUCHID is returned. The SYS$IDTOASC service returns the identifier names in alphabetical order. When SS$_NOSUCHID is returned, SYS$IDTOASC clears the context longword and deallocates the record stream. If you complete your calls to SYS$IDTOASC before SS$_NOSUCHID is returned, use SYS$FINISH_RDB to clear the context longword and to deallocate the record stream.

The following programming example uses SYS$IDTOASC to identify all identifiers in a rights database:

```
      Program ID_LIST

*
* Produce a list of all the identifiers
*

      integer SYS$IDTOASC
      external SS$_NORMAL, SS$_NOSUCHID

      character*31 NAME
      integer IDENTIFIER, ATTRIBUTES

      integer ID/-1/, LENGTH, CONTEXT/0/
      integer NAME_DSC(2)/31, 0/
      integer STATUS
*
* Initialization
*
      NAME_DSC(2) = %loc(NAME)
      STATUS = %loc(SS$_NORMAL)
*
* Scan through the entire RDB ...
*
      do while (STATUS .and. (STATUS .ne. %loc(SS$_NOSUCHID)))

         STATUS = SYS$IDTOASC(%val(ID), LENGTH, NAME_DSC,
     +                        IDENTIFIER, ATTRIBUTES, CONTEXT)

         if (STATUS .and. (STATUS .ne. %loc(SS$_NOSUCHID))) then

            NAME(LENGTH+1:LENGTH+1) = ','

            print 1, NAME, IDENTIFIER, ATTRIBUTES
    1       format(1X,'Name: ',A31,' Id: ',Z8,', Attributes: ',Z8)

         end if

      end do
*
* Do we need to finish the RDB ???
*
      if (STATUS .ne. %loc(SS$_NOSUCHID)) then
         call SYS$FINISH_RDB(CONTEXT)
      end if

      end
```

23.3.2.2 Adding Identifiers and Holders to the Rights Database

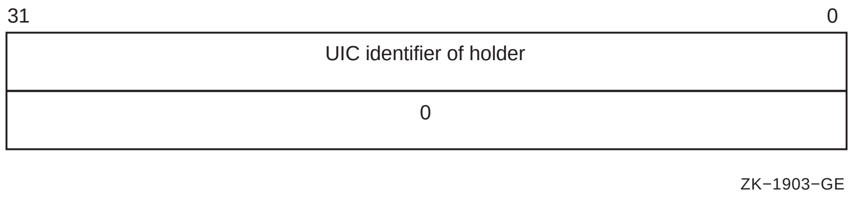
To add identifiers to the rights database, use the SYS$ADD_IDENT service in a program. When you call SYS$ADD_IDENT, use the **name** argument to pass the identifier name you want to add. You can specify an identifier value with the **id** argument; however, if you do not specify a value, the system selects an identifier value from the general identifier space.

In addition to defining the identifier value and identifier name, you use SYS$ADD_IDENT to specify attributes in the identifier record. Attributes are enabled for a holder of an identifier *only* when they are set in both the identifier record and the holder record. The **attrib** argument is a longword containing a bit mask specifying the attributes. The symbol KGB$V_RESOURCE, defined in the system macro library $KGBDEF, sets the Resource bit in the attribute longword, and the symbol KGB$V_DYNAMIC sets the Dynamic bit. (You can use the prefix KGB$M rather than KGB$V.) See the description of SYS$ADD_IDENT in the *OpenVMS System Services Reference Manual* for a complete list of symbols.

When SYS$ADD_IDENT successfully completes execution, a new identifier record containing the identifier value, the identifier name, and the attributes of the identifier exists in the rights database.

When the identifier record exists in the rights database, you define the holders of that identifier with the SYS$ADD_HOLDER system service. You pass the binary identifier value with the **id** argument and you specify the holder with the **holder** argument, which is the address of a quadword data structure in the following format. Figure 23–6 shows the format of the **holder** argument.

Figure 23–6 Format of the holder Argument



In the rights database, the holder identifier is in UIC format. You specify the attributes of the holder with the **attrib** argument in the same manner as with SYS$ADD_IDENT.

After SYS$ADD_HOLDER completes execution, a new holder record containing the binary value of the identifier that the holder holds, the attributes of the holder, and the UIC of the holder exists in the rights database.

23.3.2.3 Determining Holders of Identifiers

To determine the holders of a particular identifier, use the SYS$FIND_HOLDER service in a program. When you call SYS$FIND_HOLDER, use the **id** argument to pass the binary value of the identifier whose holder you want to determine. On successful execution, SYS$FIND_HOLDER returns the holder identifier with the **holder** argument and the attributes of the holder with the **attrib** argument.

You can identify all of the identifier's holders by initializing the **context** argument to 0 and repeatedly calling SYS$FIND_HOLDER, as detailed in Section 23.3.3. Because SYS$FIND_HOLDER identifies the records by the same key (holder ID), it returns the records in the order in which they were written.

23.3.2.4 Determining Identifiers Held by a Holder

To determine the identifiers held by a holder, use the SYS$FIND_HELD service in a program. When you call SYS$FIND_HELD, use the **holder** argument to specify the holder whose identifier is to be found.

On successful execution, SYS$FIND_HELD returns the identifier's binary identifier value and attributes.

You can identify all the identifiers held by the specified holder by initializing the **context** argument to 0 and repeatedly calling SYS$FIND_HELD, as detailed in Section 23.3.3. Because SYS$FIND_HELD identifies the records by the same key (identifier), it returns the records in the order in which they were written.

23.3.2.5 Modifying the Identifier Record

To modify an identifier record by changing the identifier's name, value, or attributes, or all three in the rights database, use the SYS$MOD_IDENT service in a program. Use the **id** argument to pass the binary value of the identifier whose record you want to modify. To enable attributes, use the **set_attrib** argument, which is a longword containing a bit mask specifying the attributes. The symbol KGB$V_RESOURCE, defined in the system macro library $KGBDEF, sets the Resource bit in the attribute longword. The symbol KGB$V_DYNAMIC sets the Dynamic bit. (You can use the prefix KGB$M rather than KGB$V.) See the description of SYS$MOD_IDENT in the *OpenVMS System Services Reference Manual* for a complete list of symbols.

If you want to disable the attributes for the identifier, use the **clr_attrib** argument, which is a longword containing a bit mask specifying the attributes. If the same attribute is specified in **set_attrib** and **clr_attrib**, the attribute is enabled.

You can also change the identifier name, value, or both with the **new_name** and **new_value** arguments. The **new_name** argument is the address of a descriptor pointing to the identifier name string; **new_value** is a longword containing the binary identifier value. If you change the value of an identifier that is the holder of other identifiers (a UIC, for example), SYS$MOD_IDENT updates all the corresponding holder records with the new holder identifier value.

When SYS$MOD_IDENT successfully completes execution, a new identifier record containing the identifier value, the identifier name, and the attributes of the identifier exists in the rights database.

23.3.2.6 Modifying a Holder Record

To modify a holder record, use the SYS$MOD_HOLDER service in a program. When you call SYS$MOD_HOLDER, use the **id** argument and the **holder** argument to pass the binary identifier value and the UIC holder identifier whose holder record you want to modify.

Use the SYS$MOD_HOLDER service to enable or disable the attributes of an identifier in the same way as with SYS$MOD_HOLDER.

When SYS$MOD_HOLDER completes execution, a new holder record containing the identifier value, the identifier name, and the attributes of the identifier exists in the rights database.

The following programming example uses SYS$MOD_HOLDER to modify holder records in the rights database:

```
      Program MOD_HOLDER

*
* Modify the attributes of all the holders of identifiers to reflect
* the current attribute setting of the identifiers themselves.
*

      external SS$_NOSUCHID
      parameter KGB$M_RESOURCE = 1, KGB$M_DYNAMIC = 2
      integer SYS$IDTOASC, SYS$FIND_HELD, SYS$MOD_HOLDER

*
* Store information about the holder here.
*

      integer HOLDER(2)/2*0/
      equivalence (HOLDER(1), HOLDER_ID)
      integer HOLDER_NAME(2)/31, 0/
      integer HOLDER_ID, HOLDER_CTX/0/
      character*31 HOLDER_STRING

*
* Store attributes here.
*

      integer OLD_ATTR, NEW_ATTR, ID_ATTR, CONTEXT

*
* Store information about the identifier here.
*

      integer IDENTIFIER, ID_NAME(2)/31, 0/
      character*31 ID_STRING
      integer STATUS
*
* Initialize the descriptors.
*

      HOLDER_NAME(2) = %loc(HOLDER_STRING)
      ID_NAME(2) = %loc(ID_STRING)

*
* Scan through all the identifiers.
*

      do while
     +   (SYS$IDTOASC(%val(-1),, HOLDER_NAME, HOLDER_ID,, HOLDER_CTX)
     +      .ne. %loc(SS$_NOSUCHID))

*
* Test all the identifiers held by this identifier (our HOLDER).
*

         if (HOLDER_ID .le. 0) go to 2

         CONTEXT = 0

         do while
     +      (SYS$FIND_HELD(HOLDER, IDENTIFIER, OLD_ATTR, CONTEXT)
     +         .ne. %loc(SS$_NOSUCHID))

*
* Get name and attributes of held identifier.
*

            STATUS = SYS$IDTOASC(%val(IDENTIFIER),, ID_NAME,, ID_ATTR,)

*
* Modify the holder record to reflect the state of the identifier itself.
*

            if ((ID_ATTR .and. KGB$M_RESOURCE) .ne. 0) then
               STATUS = SYS$MOD_HOLDER
     +                 (%val(IDENTIFIER), HOLDER, %val(KGB$M_RESOURCE),)
               NEW_ATTR = OLD_ATTR .or. KGB$M_RESOURCE
            else
               STATUS = SYS$MOD_HOLDER
     +                 (%val(IDENTIFIER), HOLDER,, %val(KGB$M_RESOURCE))
               NEW_ATTR = OLD_ATTR .and. (.not. KGB$M_RESOURCE)
            end if

            if ((ID_ATTR .and. KGB$M_DYNAMIC) .ne. 0) then
               STATUS = SYS$MOD_HOLDER
     +                 (%val(IDENTIFIER), HOLDER, %val(KGB$M_DYNAMIC),)
               NEW_ATTR = OLD_ATTR .or. KGB$M_DYNAMIC
            else
               STATUS = SYS$MOD_HOLDER
     +                 (%val(IDENTIFIER), HOLDER,, %val(KGB$M_DYNAMIC))
               NEW_ATTR = OLD_ATTR .and. (.not. KGB$M_DYNAMIC)
            end if

*
* Was it successful?
*

      if (.not. STATUS) then
         NEW_ATTR = OLD_ATTR
         call LIB$SIGNAL(%val(STATUS))
      end if
*
* Report it all.
*
```

```
          print 1, HOLDER_STRING, ID_STRING,
 +                  OLD_ATTR, ID_ATTR, NEW_ATTR
1         format(1X, 'Holder: ', A31, ' Id: ', A31,
 +                ' Old: ', Z8, ' Id: ', Z8, ' New: ', Z8)

       end do

2      continue

    end do

    end
```

23.3.2.7 Removing Identifiers and Holders from the Rights Database

To remove an identifier and all of its holders, use the SYS$REM_IDENT service in a program. When you call SYS$REM_IDENT, use the **id** argument to pass the binary value of the identifier you want to remove. When SYS$REM_IDENT completes execution, the identifier and all of its associated holder records are removed from the rights database.

To remove a holder from the list of an identifier's holders, use the SYS$REM_HOLDER service in a program. When you call SYS$REM_HOLDER, use the **id** argument and the **holder** argument to pass the binary ID value and the UIC identifier of the holder whose holder record you want to delete.

On successful execution, SYS$REM_HOLDER removes the holder from the list of the identifier's holders.

23.3.3 Search Operations

You can search the entire rights database when you use the SYS$IDTOASC, SYS$FIND_HELD, and SYS$FIND_HOLDER services. You initialize the context longword to 0 and repeatedly call one of the three services until the status code SS$_NOSUCHID is returned. When SS$_NOSUCHID is returned, the service clears the context longword and deallocates the record stream. If you complete your calls to one of these services before SS$_NOSUCHID is returned, you must use SYS$FINISH_RDB to clear the context longword and to deallocate the record stream.

The structure of the rights database affects the order in which each of these services returns the records when you search the rights database. The rights database is an indexed file with three keys. The primary key is the identifier binary value, the secondary key is the holder UIC identifier, and the tertiary key is the identifier name.

During a searching operation, the service obtains the first record with an indexed OpenVMS RMS GET operation. The key used for the GET operation depends on the service. The SYS$FIND_HOLDER service uses the identifier binary value; SYS$FIND_HELD uses the holder UIC identifier. After the indexed GET, the service returns the records with sequential RMS GET operations. Consequently, the file organization, the key used for the first GET operation, and the order in which the records were originally written in the database determine the order of records returned.

Table 23–2 summarizes how records are returned by the SYS$IDTOASC, SYS$FIND_HELD, and SYS$FIND_HOLDER services when used in a searching operation.

Table 23–2 Returned Records of SYS$IDTOASC, SYS$FIND_HELD, and SYS$FIND_HOLDER

Service	Record Order
SYS$IDTOASC	Identifier name order.
SYS$FIND_HELD	First GET operation—holder key. Subsequent records are returned in the order in which they were written.
SYS$FIND_HOLDER	First GET operation—identifier key. Subsequent records are returned in the order in which they were written.

The following programming example uses SYS$IDTOASC, SYS$FINISH_RDB, and SYS$FIND_HOLDER to search the entire rights database for identifiers with holders and produces a list of those identifiers and their holders:

```
Module ID_HOLDER
   (  main = MAIN,
      addressing_mode(external=GENERAL) ) =
begin

!
!       Produce a list of all the identifiers, that have holders,
!       with their respective holders.
!

!
!       Declarations:
!

   library

      'SYS$LIBRARY:LIB';

   forward routine

      MAIN;

   external routine

      LIB$PUT_OUTPUT,

      SYS$FAO,
      SYS$IDTOASC,
      SYS$FINISH_RDB,
      SYS$FIND_HOLDER;

!
!       To create static descriptors
!

   macro S_DESCRIPTOR[NAME, SIZE] =
      own
         %name(NAME, '_BUFFER'): block[%number(SIZE), byte],
         %name(NAME): block[DSC$K_S_BLN, byte]
                      preset( [DSC$B_CLASS] = DSC$K_CLASS_S,
                              [DSC$W_LENGTH] = %number(SIZE),
                              [DSC$A_POINTER] = %name(NAME, '_BUFFER') ); %;

!
!       Descriptors for ID, holder NAME, and output LINE
!

   S_DESCRIPTOR('ID_NAME', 31);
   S_DESCRIPTOR('NAME', 31);
   S_DESCRIPTOR('LINE', 76);

   own

      STATUS,
      ID,
      ID_LENGTH,
      ID_CONTEXT: initial(0),

      HOLDER,
      LENGTH,
      CONTEXT: initial(0),

      ATTRIBS,
      VALUE,
      LINE_: block[DSC$K_S_BLN, byte]
             preset( [DSC$B_CLASS] = DSC$K_CLASS_S,
                     [DSC$A_POINTER] = LINE_BUFFER );

!
!       To check for existence of an ID or HOLDER
!

   macro CHECK(EXPRESSION) =
      (STATUS = %remove(EXPRESSION)) and (.STATUS neq SS$_NOSUCHID) %;

!
!       List all the identifiers, which have holders, with their holders.
!

   routine MAIN =
   begin

!
!       Examine all IDs (-1).
!

      while
         CHECK(<SYS$IDTOASC(-1, ID_LENGTH, ID_NAME, ID, ATTRIBS, ID_CONTEXT)>)
      do
         begin

            CONTEXT = 0;

!
!       Find all holders of ID.
!

            while CHECK(<SYS$FIND_HOLDER(.ID, HOLDER, ATTRIBS, CONTEXT)>) do
               begin

!
!       Translate the HOLDER to find its NAME.
!

                  SYS$IDTOASC(.HOLDER, LENGTH, NAME, VALUE, ATTRIBS, 0);

!
!       Print a message reporting ID and HOLDER.
!

                  SYS$FAO( %ascid'Id: !AD, Holder: !AD',
                           LINE_[DSC$W_LENGTH], LINE,
                           .ID_LENGTH, .ID_NAME[DSC$A_POINTER],
                           .LENGTH, .NAME[DSC$A_POINTER] );

                  LIB$PUT_OUTPUT(LINE_);

               end;

         end;

      return SS$_NORMAL;

   end;

end

eludom
```

23.3.4 Modifying a Rights List

When a process is created, LOGINOUT builds a rights list for the process consisting of the identifiers the user holds and any appropriate environmental identifiers. A **system rights list** is the default rights list used in addition to any process rights list. Modifications to the system rights list effectively become modifications to the rights of each process.

A privileged user can alter the process or system rights list with the SYS$GRANTID or SYS$REVOKID services. These services are not intended for the general system user. Use of these services requires CMKRNL privilege. The SYS$GRANTID service adds an identifier to a rights list or, if the identifier is already part of the rights list, the SYS$GRANTID service modifies the attributes of the identifier. The SYS$REVOKID service removes an identifier from a rights list.

The SYS$GRANTID and SYS$REVOKID services treat the **pidadr** and **prcnam** arguments the same way all other process control services treat these arguments. For more details, see the *OpenVMS Guide to System Security*.

You can also modify the process or system rights list with the DCL command SET RIGHTS_LIST. Additionally, you can use SET RIGHTS_LIST to modify the attributes of the identifier if the identifier is already part of the rights list. Note that you cannot use the SET RIGHTS_LIST command to modify the rights database from which the rights list was created. For more information about using the SET RIGHTS_LIST command, see the *OpenVMS DCL Dictionary*.

23.4 Managing Object Protection

An ACL is a list of entries defining the type of access allowed to an object in the system such as a file, device, or mailbox. An access control entry (ACE) consists of an identifier and one or more access types.

```
(IDENTIFIER=GREEN,ACCESS=WRITE+READ+CONTROL)
(IDENTIFIER=YELLOW,ACCESS=READ)
(IDENTIFIER=RED,ACCESS=NOACCESS)
```

Managing object protection involves using system services to manipulate protection codes, UICs, and ACEs; that is, creating, translating, and maintaining ACEs, establishing object ownership, and manipulating the protection codes of protected objects.

23.4.1 Protected Objects

A **protected object** is an entity that can contain or receive information. When such information is not considered shareable, access to those objects can be restricted. The system recognizes eleven classes of protected objects as shown in the following table:

Class Name	Description
Capability[1]	A resource to which the system controls access; currently, the only defined capability is the vector processor.
Common event flag cluster	A set of 32 event flags that enable cooperating processes to post event notifications to each other.

[1]Exists only on systems with vector processors

Class Name	Description
Device	A class of peripherals connected to a processor that are capable of receiving, storing, or transmitting data.
File	Files–11 On-Disk Structure Level 2 (ODS-2) files and directories.
Group global section	A shareable memory section potentially available to all processes in the same group.
Logical name table	A shareable table of logical names and their equivalence names for the system or a particular group.
Queue	A set of jobs to be processed in a batch, terminal, server, or print job queue.
Resource domain	A namespace controlling access to the lock manager's resources.
Security class	A data structure containing the elements and management routines for all members of the security class.
System global section	A shareable memory section potentially available to all processes in the system.
Volume	A mass storage medium, such as a disk or tape, that is in ODS-2 format. Volumes contain files and may be mounted on devices.

23.4.2 Object Security Profile

The security profile summarizes the various types of protection mechanisms applied to a protected object. The security profile associates a protected object with an owner, a protection code, and optionally an ACL. When a user or process requests access to a protected object, the system compares the user's privileges and identifiers in the system authorization database with appropriate elements in the object's security profile.

23.4.2.1 Displaying the Security Profile

You can display an object's security profile by using the SYS$GET_SECURITY system service. On your first call to SYS$GET_SECURITY, be sure to initialize the context variable to 0. Use the OSS$M_RELCTX flag to release any locks on the context structure when the routine completes execution. The following example illustrates the type of information contained in the security profile of a logical name table:

```
LNM$GROUP object of class LOGICAL_NAME_TABLE

        Owner: [ACCOUNTING]
        Protection: (System: RWCD, Owner: RWCD, Group: R, World: R)
        Access Control List:
                (IDENTIFIER=[USER,CHEHKOV],ACCESS=CONTROL)
                (IDENTIFIER=[USER,VANNEST],ACCESS=READ+WRITE)
```

After you have returned owner and protection code information, you can call SYS$GET_SECURITY iteratively to return each ACE in the ACL (if it exists) or you can read the entire ACL. In addition, you can perform iterative searches to retrieve objects and their templates.

23.4.2.2 Modifying the Security Profile

You can modify all the security characteristics listed in a protected object's profile by using the SYS$SET_SECURITY system service. You can add or delete ACEs in the ACL selectively or you can delete the entire ACL. You have the option of modifying a local copy of the profile without altering the master copy using the OSS$M_LOCAL flags or you can modify the master copy directly. Also, use the context to release the context structure after the service completes execution.

23.4.3 Types of Access Control Entries

There are seven types of security-related ACEs as described in the following table:

ACE	Description
Alarm	Sets an alarm
Application	Contains application-dependent information
Audit	Sets a security audit
Creator	Controls access to an object based on creators
Default Protection	Specifies the default protection for all files and subdirectories created in the directory
Identifier	Controls the type of access allowed based on identifiers
Subsystem	Maintains protected subsystems

For information about the structure of specific types of ACEs, see the SYS$FORMAT_ACL system service in *OpenVMS System Services Reference Manual*.

You use SYS$FORMAT_ACL and SYS$PARSE_ACL to translate ACEs from one format to another in the same way that SYS$IDTOASC and SYS$ASCTOID translate identifiers from binary to text format and text to binary format.

To create and manipulate ACLs, use the ACL editor, the DCL command SET ACL, or the SYS$GET_SECURITY and SYS$SET_SECURITY system services in a program. The following table lists services that manipulate ACEs:

Service	Description
SYS$FORMAT_ACL	Converts an ACE from binary format to ASCII text
SYS$GET_SECURITY	Retrieves the security characteristics of an object
SYS$PARSE_ACL	Converts an ACE from ASCII text to binary format
SYS$SET_SECURITY	Modifies the security characteristics of a protected object

23.4.3.1 Design Considerations

Before you attempt to manipulate ACLs, you should understand the meaning and relationship among existing identifiers. If you are populating a previously empty ACL, you need to plan the access types and position of each ACE within the ACL.

The position of the ACE within the ACL is an important consideration when creating an ACE. By default, ACEs are added to the top of an ACL. The ACL management services accept options allowing you to control the placement of ACEs. The system compares the identifiers granted to the process requesting access with those associated with the object starting with the top ACE in the object's ACL. Once a matching identifier name is found in the object's ACL, the search stops.

23.4.3.2 Translating ACEs

To translate ACEs from binary format to a text string, use the SYS$FORMAT_ACL service. The **aclent** argument is the address of a descriptor pointing to a buffer containing the description of the ACE. The first byte of the buffer contains the length of the ACE and the second byte contains the type, which in turn defines the format of the ACE.

The **acllen** argument specifies the length of the text string written to the buffer pointed to by **aclstr**. You use the **width**, **trmdsc**, and **indent** arguments to specify a particular width, termination character, and number of blank characters for an ACE. The **accnam** argument contains the address of an array of 32 quadword descriptors called an **access name table**. The access name table defines the names of the bits in the access mask of the ACE. The access mask defines the access types associated with a protected object. Use run-time library (RTL) routine LIB$GET_ACCNAM described in the *OpenVMS RTL Library (LIB$) Manual* to obtain the address of the access name table. If **accnam** is omitted, the following names are used:

```
Bit <0>    READ
Bit <1>    WRITE
Bit <2>    EXECUTE
Bit <3>    DELETE
Bit <4>    CONTROL
Bit <5>    BIT_5
Bit <6>    BIT_6
   .
   .
   .
Bit <31>   BIT_31
```

The SYS$PARSE_ACL service translates an ACE from text string format to binary format. The **aclstr** argument is the address of a string descriptor pointing to the ACE text string. As with SYS$FORMAT_ACL, the **aclent** argument is the address of a descriptor pointing to a buffer containing the description of the ACE. The first byte of the buffer contains the length of the ACE and the second byte contains the type, which in turn defines the format of the ACE. If SYS$PARSE_ACL fails, the **errpos** argument points to the failing point in the string. The **accnam** argument contains the address of an array of 32 quadword descriptors that define the names of the bits in the access mask of the ACE. If **accnam** is omitted, the names specified in the description of SYS$FORMAT_ACL are used.

23.4.3.3 Creating and Maintaining ACEs

The SYS$GET_SECURITY and SYS$SET_SECURITY system services replace the SYS$CHANGE_ACL system service. The *OpenVMS System Services Reference Manual: A–GETMSG* and the *OpenVMS System Services Reference Manual: GETQUI–Z* describe these system services.

To create or modify an ACL associated with a protected object, you use the SYS$SET_SECURITY service. You specify the object whose ACL is to be modified with either the **objhan** argument, which specifies the I/O channel associated with the object, or the **objnam** argument, which specifies the object name. If you specify **objnam**, **objhan** must be omitted or specified as 0. The **clsnam** argument specifies the type of object.

Use the **acmode** argument to specify the access mode used when checking file access protection. By default, kernel mode is used, but the system compares **acmode** against the caller's access mode and uses the least privileged mode. Digital recommends that this argument be omitted (passed as zero).

The item code specifies the change to be made to the ACL. Table 23–3 describes the symbols for the item codes that are defined in the system macro library ($ACLDEF). Note that without the **itmlst** argument, you can manipulate only the security profile locks or release **contxt** resources.

Table 23–3 Item Code Symbols and Meanings

Item Code	Description
OSS$_ACL_ADD_ENTRY	Adds an access control entry (ACE)
OSS$_ACL_DELETE	Deletes all unprotected ACEs from an ACL
OSS$_ACL_DELETE_ALL	Deletes the ACL, including protected ACEs
OSS$_ACL_DELETE_ENTRY	Deletes an ACE
OSS$_ACL_FIND_ENTRY	Locates an ACE
OSS$_ACL_FIND_NEXT	Moves the current position to the next ACE in the ACL
OSS$_ACL_FIND_TYPE	Locates an ACE of the specified type
OSS$_ACL_MODIFY_ENTRY	Replaces an ACE at the current position
OSS$_ACL_POSITION_BOTTOM	Sets a marker that points to the end of the ACL
OSS$_ACL_POSITION_TOP	Sets a marker that points to the beginning of the ACL
OSS$_OWNER	Sets the UIC or general identifier of the object's owner
OSS$_PROTECTION	Sets the protection code of the object

23.5 Protected Subsystems

A protected subsystem is a set of application programs that allow controlled access to objects. It has under its control one or more protected objects and a gatekeeper application. Users cannot access the objects within the subsystem unless they execute the gatekeeper application. Once users have successfully executed the application, their process rights list acquires the identifiers necessary to access objects owned by the subsystem. The identifiers allow processes to use the resources of the subsystem. When the application completes execution or the user exits, the identifiers are removed from the user's process rights list. Protected subsystems are an alternative to creating privileged images and protected shareable images (user-written system services), and help prevent the overuse of privileges.

Roles and Responsibilities

You should think of a protected subsystem as an isolated security domain where the system manager creates and grants SUBSYSTEM identifiers using the Authorize utility as shown in the following example:

```
UAF> ADD/IDENTIFIER FOO/ATTRIBUTES=SUBSYSTEM
UAF> GRANT/IDENTIFIER FOO FRANK /ATTRIBUTES=SUBSYSTEM
```

The system manager can delegate responsibility for the maintenance of the subsystem to subsystem managers who can associate existing identifiers with the subsystem executable and its data. In the following example, the manager of a protected subsystem creates an ACE in a subsystem's image and data files:

```
$ SET SECURITY BLOP.EXE -
_$ /ACL=(SUBSYSTEM, IDENTIFIER=FOO) -
$ SET SECURITY BLOP.DAT -
_$ /ACL=(IDENTIFIER=FOO, ACCESS=READ+WRITE) -
$ SET SECURITY BLOP.EXE -
_$ /ACL=(IDENTIFIER=HARRY, ACCESS=EXECUTE) -
```

Finally, a user uses the protected subsystem to access data available only through the subsystem.

Subsystem Security

During the execution of a protected subsystem, $IMGACT adds subsystem identifiers to the image rights list. What happens if the user presses the Ctrl/Y key sequence during execution? Will the user retain whatever privileges were granted by the subsystem? If Ctrl/Y is pressed, image identifiers are removed from the process. Also, subprocesses do not inherit image identifiers by default. However, SYS$CREPRC and LIB$SPAWN do contain flags PRC$M_SUBSYSTEM and SUBSYSTEM, respectively, that allow subprocesses to inherit image identifiers.

23.6 Security Auditing

Auditing is the recording of security-relevant activity as it occurs on a system. See the *OpenVMS Guide to System Security* for a list of all types of security-relevant activity or classes of **events** that are audited. The following table describes the security services that provide security auditing:

Service	Description
SYS$AUDIT_EVENT	Appends an event message to the system audit log file or sends an alarm to a security operator terminal
SYS$CHECK_PRIVILEGE	Determines whether the caller has the specified privileges or identifiers

The system service SYS$AUDIT_EVENT is used to report security events to the auditing system. It examines the settings of the DCL command SET AUDIT to determine if an event is enabled for auditing. If the event is enabled for alarms or audits, SYS$AUDIT_EVENT generates an audit record and appends it to the system audit log file (or sends an alarm to a security operator terminal) that identifies the process involved and lists information supplied by its caller.

23.7 Checking Access Protection

The operating system provides two system services that allow a process to check access to objects on the system: SYS$CHKPRO and SYS$CHECK_ACCESS. The SYS$CHKPRO service performs the system access protection check on a user attempting direct access to an object on the system; SYS$CHECK_ACCESS performs a similar check on a third party attempting access to an object. The following table describes the security services that provide access checking:

Service	Description
SYS$CHECK_ACCESS	Invokes a system access protection check on behalf of another user
SYS$CHKPRO	Invokes a system access protection check

The SYS$CHKPRO and SYS$CHECK_ACCESS system services have been extended to support auditing. The *OpenVMS Guide to System Security* describes how to use the auditing function. The *OpenVMS System Services Reference Manual: A–GETMSG* describes how to use the two system services. These services are described in the following sections.

23.7.1 Creating a Security Profile

The SYS$CREATE_USER_PROFILE system service returns a user profile, using information in the rights database and the system authorization database to generate the profile. The system services SYS$CHECK_ACCESS or SYS$CHKPRO accept as input the profile from SYS$CREATE_USER_PROFILE.

23.7.2 SYS$CHKPRO System Sevice

The SYS$CHKPRO service invokes the access protection check used by the system. The service does not grant or deny access; rather, it performs the protection check. Subsequently, an application might grant or deny access to the specified object.

To pass the input and output information to SYS$CHKPRO, use the **itmlst** argument, which is the address of an item list of descriptors. The SYS$CHKPRO service compares the item list of the rights and privileges of the accessor to a list of the protection attributes of the object to be accessed. If the accessor can access the object, SYS$CHKPRO returns the status SS$_NORMAL; if the accessor cannot access the object, SYS$CHKPRO returns the status SS$_NOPRIV. The SYS$CHKPRO service does not grant or deny access. The subsystem itself must grant or deny access based on the output (SS$_NORMAL or SS$_NOPRIV) from SYS$CHKPRO.

The SYS$CHKPRO service also returns an item list of the rights or privileges that allowed the accessor access to the object, as well as the names of security alarms raised by the access attempt. For information about the item codes defined for SYS$CHKPRO, see the description of SYS$CHKPRO in the *OpenVMS System Services Reference Manual*.

See the *OpenVMS Guide to System Security* for a flowchart describing how SYS$CHKPRO evaluates an access request attempt.

23.7.3 SYS$CHECK_ACCESS System Service

The SYS$CHECK_ACCESS service performs a protection check on a third-party accessor. An example of this is a file server program that uses SYS$CHECK_ACCESS to ensure that an accessor (the third party) requesting a file has the required privileges to do so.

You pass the input and output information to SYS$CHECK_ACCESS by using the **itmlst** argument, which is the address of an item list of descriptors. You also pass the name of the accessor and the name and type of the object being accessed by using the **usrnam**, **objnam**, and **objtyp** arguments, respectively. The SYS$CHECK_ACCESS service compares the rights and privileges of the accessor to a list of the protection attributes of the object to be accessed. If the accessor can access the object, SYS$CHECK_ACCESS returns the status

SS$_NORMAL; if the accessor cannot access the object, SYS$CHECK_ACCESS returns the status SS$_NOPRIV.

The SYS$CHECK_ACCESS service does not grant or deny access. The subsystem itself must explicitly grant or deny access based on the output (SS$_NORMAL or SS$_NOPRIV) from SYS$CHECK_ACCESS.

The SYS$CHECK_ACCESS service also returns an item list of the rights or privileges that allowed the accessor to access the object, as well as the names of security alarms raised by the access attempt. For information about the item codes defined for SYS$CHECK_ACCESS, see the description of SYS$CHECK_ACCESS in the *OpenVMS System Services Reference Manual*.

23.8 SYS$CHECK_PRIVILEGE

The SYS$CHECK_PRIVILEGE system service determines whether the caller has the specified privileges or identifiers. The service performs the privilege check and looks at the SET AUDIT settings to determine whether the system administrator enabled privilege auditing. When privilege auditing is enabled, SYS$CHECK_PRIVILEGE generates an audit record. The audit record identifies the process (subject) and privilege involved, provides the result of the privilege check, and lists supplemental event information supplied by its caller. Privilege audit records usually contain the DCL command line or the system service name associated with the privilege check.

SYS$CHECK_PRIVILEGE completes asynchronously; that is, it does not wait for final status. For synchronous completion, use the SYS$CHECK_PRIVILEGEW service.

23.9 Implementing Site-Specific Security Policies (VAX Only)

Occasionally, you may need to write routines that implement site-specific policies or special algorithms. The routines that you write can either replace or augment built-in operating system policies. This section contains instructions for replacing key operating system security routines with routines that are specific to your site. Two types of routines are discussed: loadable system services and shareable images.

23.9.1 Creating Loadable Security Services

This section describes how to create a system service image and how to update the SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA file, which controls site-specific loading of system images. These procedures update the loading of system images for all nodes of a cluster.

Currently, you can replace the following three system services with services specific to your site:

Service	Description
SYS$ERAPAT	Generates a security erasure pattern
SYS$MTACCESS	Controls magnetic tape access
SYS$HASH_PASSWORD	Applies a hash algorithm to an ASCII password

When you create the system service, you code the source module and define the vector offsets, the entry point, and the program sections for the system service. Then, you can assemble and link the module to create a loadable image.

Once you have created the loadable image, you install it. First, you copy the image into the SYS$LOADABLE_IMAGES directory and add an entry for it in the operating system’s images file using the System Management utility (SYSMAN). Next, you invoke the system images command procedure to generate a new system image data file. Finally, you reboot the system to load your service.

The following sections describe how to create and load the the Get Security Erase Pattern (SYS$ERAPAT) system service.

On VAX systems, you can find an example of the SYS$ERAPAT system service in SYS$EXAMPLES:DOD_ERAPAT.MAR on the operating system. The description here also applies to the Hash Password (SYS$HASH_PASSWORD) and Magnetic Tape Accessibility (SYS$MTACCESS) system services. You can find an example of how to prepare and load the SYS$HASH_PASSWORD service in SYS$EXAMPLES:HASH_PASSWORD.MAR.

23.9.1.1 Preparing and Loading a System Service

On VAX systems, use the following procedure to prepare and load a system service, in this case SYS$ERAPAT:

1. Create the source module.
   a. Include the following macro to define system service vector offsets:
   ```
   $SYSVECTORDEF   ; Define system service vector offsets
   ```
   b. Use the following macro to define the system service entry point:
   ```
   SYSTEM_SERVICE ERAPAT, -     ; Entry point name
                  <R4>, -       ; Register to save
                  MODE=KERNEL,-   ; Mode of system service
                  NARG=3          ; Number of arguments
   ```
   (The code immediately following this macro is the first instruction of the SYS$ERAPAT system service.)
   c. Use the following macros to declare the desired program sections:
   ```
   DECLARE_PSECT   EXEC$PAGED_CODE ; Pageable code PSCET

   DECLARE_PSECT   EXEC$PAGED_DATA ; Pageable data PSECT

   DECLARE_PSECT   EXEC$NONPAGED_DATA   ; Nonpageable data PSECT

   DECLARE_PSECT   EXEC$NONPAGED_CODE   ; Nonpageable code PSCET
   ```
2. Assemble the source module by using the following command:
   ```
   $ MACRO DOD_ERAPAT+SYS$LIBRARY:LIB.MLB/LIB
   ```
3. Link the module to create a SYS$ERAPAT.EXE executive loaded image. You can link the module using the command procedure DOD_ERAPAT_LNK.COM in SYS$EXAMPLES. (A command procedure is also available to link the SYS$HASH_PASSWORD example.) To link the SYS$ERAPAT module, enter the following command:
   ```
   $ @SYS$EXAMPLES:DOD_ERAPAT_LNK.COM
   ```
4. Prepare the operating system image to be loaded.
   a. Copy the SYS$ERAPAT.EXE image produced by the link command into the SYS$COMMON:[SYS$LDR] directory. Note that privilege is required to put files into this directory.
   b. Add an entry for the SYS$ERAPAT.EXE image in the SYS$UPDATE:VMS$SYSTEM_IMAGES.IDX data file.

You add an entry by using the SYSMAN command SYS_LOADABLE ADD. (See the *OpenVMS System Management Utilities Reference Manual* for a description of this command.) For example, the following commands add an entry in VMS$SYSTEM_IMAGES.IDX for SYS$ERAPAT.EXE:

```
$ RUN SYS$SYSTEM:SYSMAN
SYSMAN> SYS_LOADABLE ADD _LOCAL_ SYS$ERAPAT -
_SYSMAN> /LOAD_STEP = SYSINIT -
_SYSMAN> /SEVERITY = WARNING -
_SYSMAN> /MESSAGE = "failure to load SYS$ERAPAT.EXE"
```

This entry specifies that the SYS$ERAPAT.EXE image is to be loaded by the SYSINIT process during the bootstrap. If there is an error loading the image, the following messages are printed on the console terminal:

```
%SYSINIT-E-failure to load SYS$ERAPAT.EXE
-SYSINIT-E-error loading <SYS$LDR>SYS$ERAPAT.EXE, status = "status"
```

The following table shows other error messages that may be returned:

Message	Meaning	User Action
NO_PHYSICAL_ MEMORY	Physical memory is not available.	Check SYSGEN parameters.
NO_POOL	Amount of nonpaged pool is insufficient.	Check SYSGEN parameters.
MULTIPLE_ISDS	Encountered more than one image section of a given type.	Check link options.
BAD_GSD	An inconsistency was detected.	Verify that the image was linked properly.
NO_SUCH_IMAGE	The requested image cannot be located.	Check image name against images in SYS$LOADABLE_ IMAGES.

c. Invoke the SYS$UPDATE:VMS$SYSTEM_IMAGES.COM command procedure to generate a new system image data file (VMS$SYSTEM_ IMAGES.DATA). The system bootstrap uses this image data file to load the appropriate images into the system.

d. Reboot the system, which loads the original SYS$ERAPAT.EXE image into the system. Subsequent calls to the SYS$ERAPAT system service use the normal operating system routine.

   As the default, the system bootstrap loads all images described in VMS$SYSTEM_IMAGES.DATA. You can disable this feature by setting the special system parameter LOAD_SYS_IMAGES to 0.

23.9.1.2 Removing an Executive Loaded Image

On VAX systems, use the following procedure to remove an executive loaded image; in this case, SYS$ERAPAT.EXE:

1. Enter the following SYSMAN command:

```
SYSMAN> SYS_LOADABLE REMOVE _LOCAL_ SYS$ERAPAT
```

2. Invoke the SYS$UPDATE:VMS$SYSTEM_IMAGES.COM command procedure to generate a new system image data file (VMS$SYSTEM_ IMAGES.DATA). The system bootstrap uses this image data file to load the appropriate images into the system.

3. Reboot the system, which loads the installation-specific SYS$ERAPAT.EXE image into the system. Subsequent calls to the SYS$ERAPAT system service use the installation-specific routine.

   As the default, the system bootstrap loads all images described in the system image data file (VMS$SYSTEM_IMAGES.DATA). You can disable this functionality by setting the special system parameter LOAD_SYS_IMAGES to 0.

23.9.2 Installing Filters for Site-Specific Password Policies

A site security administrator can screen new passwords to make sure they comply with a site-specific password policy. (See the *OpenVMS Guide to System Security* for more information.) This section describes how a security administrator can encode the policy, create a shareable image and install it in SYS$LIBRARY, and enable the policy by setting a SYSGEN parameter.

Installing and enabling a site-specific password policy image requires both SYSPRV and CMKRNL privileges.

23.9.2.1 Creating a Shareable Image

To compile and link a shareable image that filters passwords for words that are sensitive to your site, perform the following steps:

1. Create the source module VMS$PASSWORD_POLICY.*.

   Bliss and Ada examples of the policy module's interface, called VMS$PASSWORD_POLICY.*, are located in SYS$EXAMPLES.

   Define two routine names in the source module: POLICY_PLAINTEXT and POLICY_HASH. These routines must be global (see your language reference for instructions on defining a global routine). The Set Password utility looks for these routine names and displays the message SYMNOTFOU if the names are missing or if the routines are not defined as global.

2. Link the source file.

   On VAX systems, use the VMS$PASSWORD_POLICY_LNK.COM command procedure, located in SYS$EXAMPLES.

23.9.2.2 Installing a Shareable Image

To install a shareable image, perform the following steps:

1. Copy the file to SYS$LIBRARY and install it using the following commands:

```
$ COPY VMS$PASSWORD_POLICY.EXE SYS$COMMON:[SYSLIB]/PROTECTION=(W:RE)
$ INSTALL ADD SYS$LIBRARY:VMS$PASSWORD_POLICY/OPEN/HEAD/SHARE
```

2. Set the system parameter LOAD_PWD_POLICY to 1 as follows:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> USE ACTIVE
SYSGEN> SET LOAD_PWD_POLICY 1
SYSGEN> WRITE ACTIVE
SYSGEN> WRITE CURRENT
```

3. To make the changes permanent, add the INSTALL command from step 1 to the SYS$SYSTEM:SYSTARTUP_VMS.COM file and modify the system parameter file, MODPARAMS.DAT, so that the LOAD_PWD_POLICY parameter is set to 1.
4. Run AUTOGEN as follows to ensure that the system parameters are set correctly on subsequent system startups:

   ```
   $ @SYS$UPDATE:AUTOGEN SAVPARAMS SETPARAMS
   ```

Index